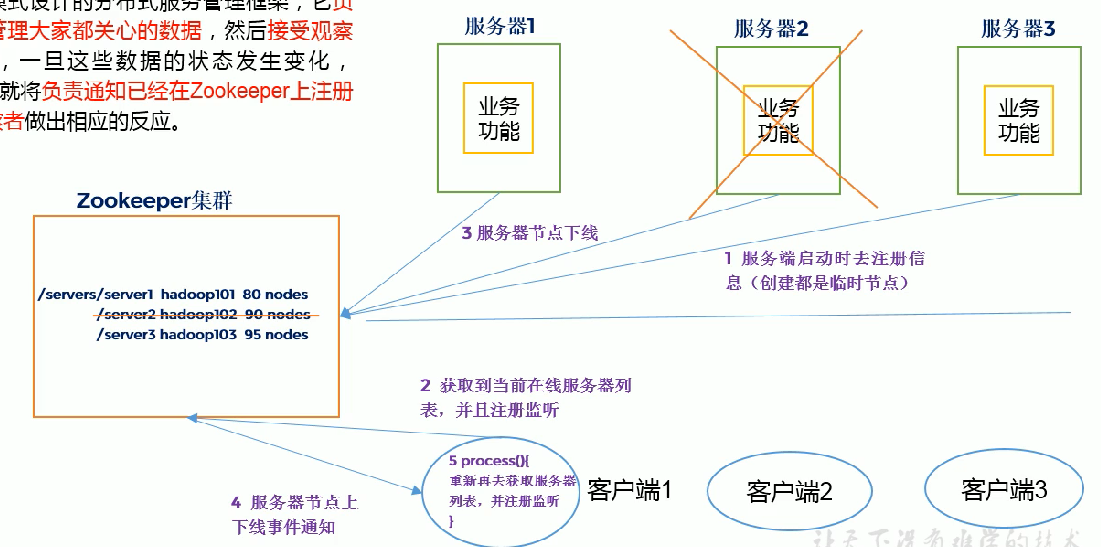
# Zookeeper

## Zookeeper

### Zookeeper简介

* 

Zookeeper是一个开源的分布式的，为分布式应用提供协调服务的Apache项目。

Zookeeper从设计模式角度来理解：是一个基于观察者模式设计的分布式服务管理框架，

它负责存储和管理大家都关心的数据，然后接受观察者的注册，一旦这些数据的状态发生变化，Zookeeper 就将负责通知已经在Zookeeper上注册的那些观察者做出相应的反应，从而实现集群中类似Master/Slave管理模式

Zookeeper=文件系统+通知机制

文件系统

Zookeeper维护了一个类似文件系统的数据结构：每个子目录项如 NameService 都被称作为 znode，和文件系统一样，我们能够自由的增加、删除znode，在一个znode下增加、删除子znode，唯一的不同在于znode是可以存储数据的。

通知机制

客户端注册监听它关心的目录节点，当目录节点发生变化（数据改变、被删除、子目录节点增加删除）时，zookeeper会通知客户端。

### ZooKeeper 的特性

**顺序一致性**：所有客户端看到的服务端数据模型都是一致的；从一个客户端发起的事务请求，最终都会严格按照其发起顺序被应用到 ZooKeeper 中。具体的实现可见下文：原子广播。

**原子性**：所有事务请求的处理结果在整个集群中所有机器上的应用情况是一致的，即整个集群要么都成功应用了某个事务，要么都没有应用。实现方式可见下文：事务。

**单一视图**：无论客户端连接的是哪个 Zookeeper 服务器，其看到的服务端数据模型都是一致的。

**高性能**：ZooKeeper 将数据全量存储在内存中，所以其性能很高。需要注意的是：由于 ZooKeeper 的所有更新和删除都是基于事务的，因此 ZooKeeper 在读多写少的应用场景中有性能表现较好，如果写操作频繁，性能会大大下滑。

**高可用**：ZooKeeper 的高可用是基于副本机制实现的，此外 ZooKeeper 支持故障恢复，可见下文：选举 Leader

### ZooKeeper 的设计目标

简单的数据模型

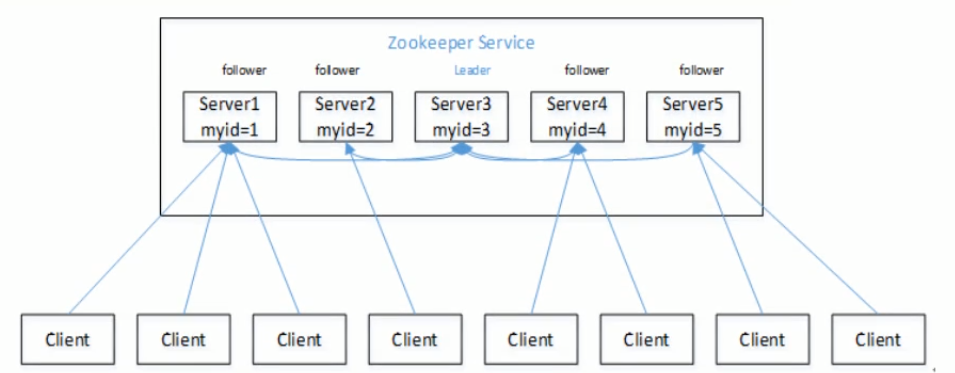
可以构建集群

顺序访问

高性能

## Zookeeper集群

### 集群



1）Zookeeper：一个领导者（leader)，多个跟随者（follower)组成的集群。

2）Leader负责进行投票的发起和决议，更新系统状态

3）Follower 用于接收客户请求并向客户端返回结果，在选举Leader过程中参与投票

4）集群中只要有半数以上节点存活，Zookeeper集群就能正常服务。

5）全局数据一致：每个server保存一份相同的数据副本，client无论连接到哪个server，数据都是一致的。

6）更新请求顺序进行，来自同一个client的更新请求按其发送顺序依次执行。

7）数据更新原子性，一次数据更新要么成功，要么失败。

8）实时性，在一定时间范围内，client 能读到最新数据。

9)集群中只要有半数以上节点存活，Zookeeper集群就能正常服务。

## 数据模型

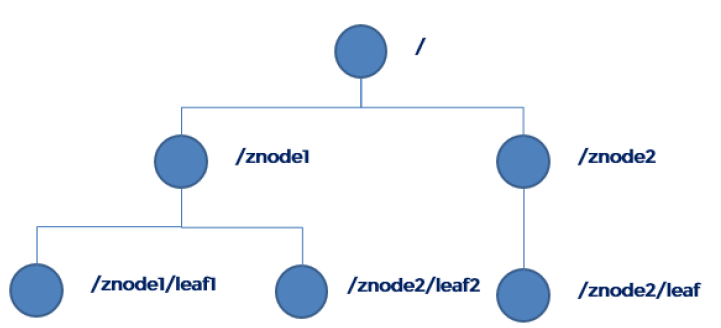
### Znode

Zookeeper的数据模型是层次模型(GoogleChubby也是这么做的)。层次模型常见于文件系统。层次模型和key-value模型是两种主流的数据模型。ZooKeeper使用文件系统模型主要基于以下两点考虑:

1、文件系统的树形结构便于表达数据之间的层次关系。

2、文件系统的树形结构便于为不同的应用分配独立的命名空间( namespace )。

很显然zookeeper集群自身维护了一套数据结构。这个存储结构是一个树形结构，其上的每一个节点，我们称之为“znode”，每一个znode默认能够存储1MB的数据，每个ZNode都可以通过其路径唯一标识



每个子目录项如 NameService 都被称作为 znode，和文件系统一样，我们能够自由的增加、删除znode，在一个znode下增加、删除子znode，唯一的不同在于znode是可以存储数据的。

ZooKeeper 的数据访问具有原子性。其读写操作都是要么全部成功，要么全部失败。

### znode类型

有四种类型的znode：

1、PERSISTENT-持久化目录节点

客户端与zookeeper断开连接后，该节点依旧存在

2、 PERSISTENT\_SEQUENTIAL-持久化顺序编号目录节点

客户端与zookeeper断开连接后，该节点依旧存在，只是Zookeeper给该节点名称进行顺序编号

3、EPHEMERAL-临时目录节点

客户端与zookeeper断开连接后，该节点被删除

4、EPHEMERAL\_SEQUENTIAL-临时顺序编号目录节点

客户端与zookeeper断开连接后，该节点被删除，只是Zookeeper给该节点名称进行顺序编号

注意：在分布式系统中，顺序号可以被用于为所有的事件进行全局排序，这样客户端可以通过顺序号推断事件的顺序

### znode信息

znode 上有一个顺序标志（ SEQUENTIAL ）。如果在创建 znode 时，设置了顺序标志（ SEQUENTIAL ），那么 ZooKeeper 会使用计数器为 znode 添加一个单调递增的数值，即 zxid。ZooKeeper 正是利用 zxid 实现了严格的顺序访问控制能力。

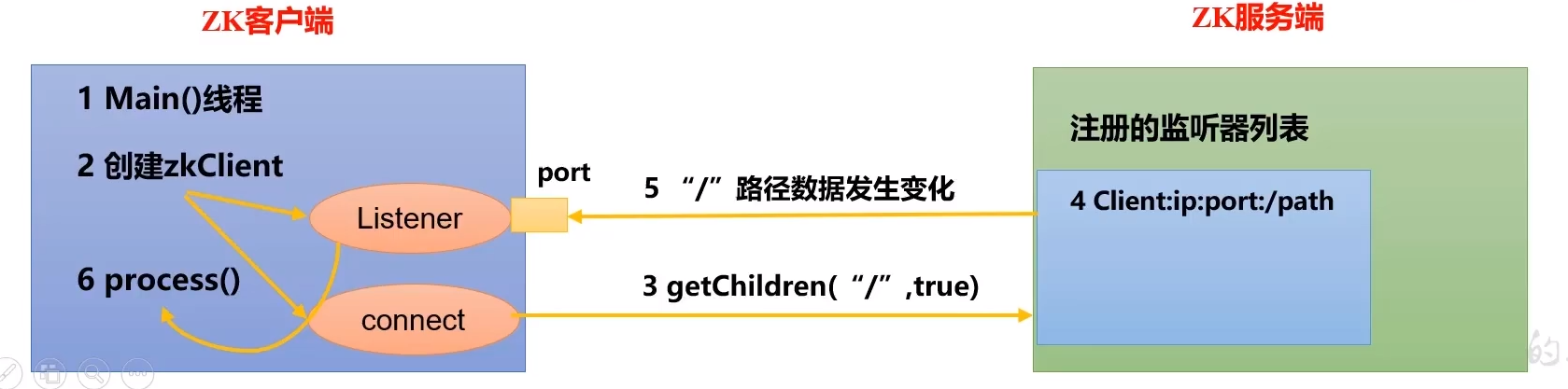
每个 znode 节点在存储数据的同时，都会维护一个叫做 Stat 的数据结构，里面存储了关于该节点的全部状态信息。如下：



## 结点的监听

### 监听原理

客户端注册监听它关心的目录节点，当目录节点发生变化（数据改变、节点删除、子目录节点增加删除）时，ZooKeeper 会通知客户端。监听机制保证 ZooKeeper 保存的任何的数据的任何改变都能快速的响应到监听了该节点的应用程序。



监听原理详解

1）首先要有一个main()线程

2）在main线程中创建Zookeeper客户端，这时就会创建两个线程，一个负责网络连接通信（connet），一个负责监听（listener）。

3）通过connect线程将注册的监听事件发送给Zookeeper。

4）在Zookeeper的注册监听器列表中将注册的监听事件添加到列表中。5）Zookeeper监听到有数据或路径变化，就会将这个消息发送给listener线程。

6）listener线程内部调用了process()方法。

常见的监听:

1）监听节点数据的变化

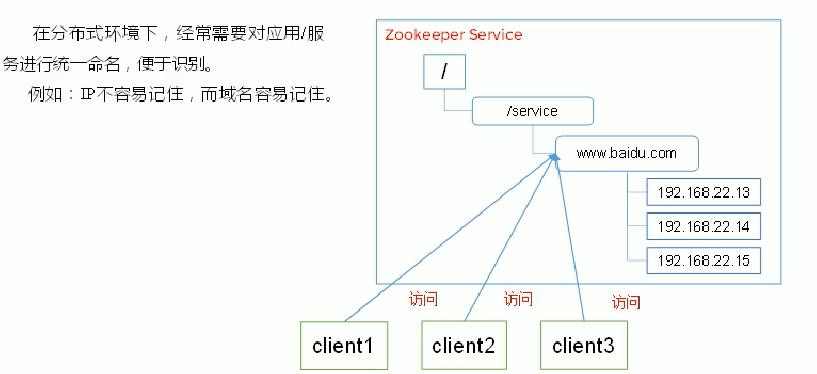
get path [watch]

2、监听子节点增减的变化

ls path [watch]

## zooker的应用场景

提供的服务包括：分布式消息同步和协调机制、服务器节点动态上下线、统一配置管理、负载均衡、集群管理等。



1、统一命名服务

在分布式环境下，经常需要对应用/服务进行统一命名，便于识别。

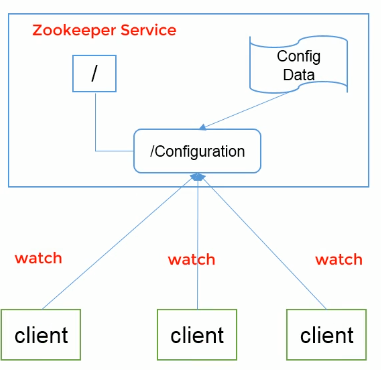
例如：IP不容易记住，而域名容易记住。

2、统一配置管理

（1)一般要求一个集群中，所有节点的配置信息是一致的，比如Kafka集群。

（2）对配置文件修改后，希望能够快速同步到各个节点上。

3、配置管理可交由ZooKeeper实现。



1）可将配置信息写入ZooKeeper上的一个Znode。

2）各个客户端服务器监听这个Znode。

(3)一旦Znode中的数据被修改，ZooKeeper将通知各个客户端服务器。

4、统一集群的管理

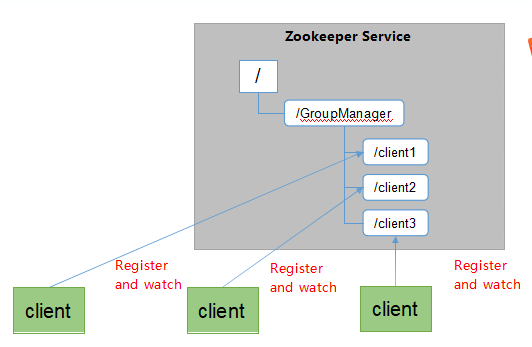
1、分布式环境中，实时掌握每个节点的状态是必要的。

（1）可根据节点实时状态做出一些调整。

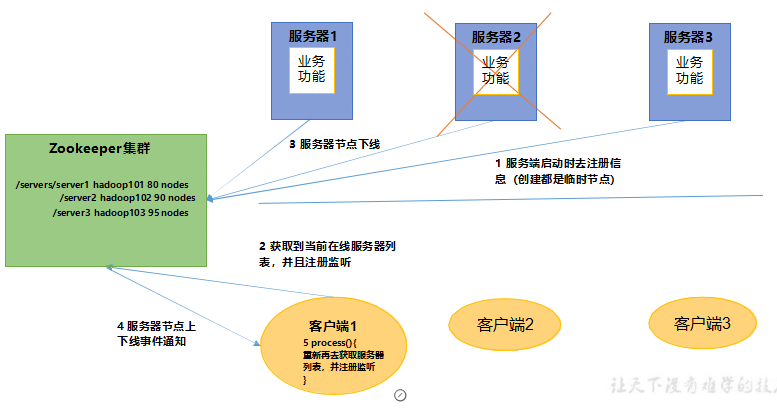
2、ZooKeeper可以实现实时监控节点状态变化

（1）可将节点信息写入ZooKeeper上的一个ZNode。

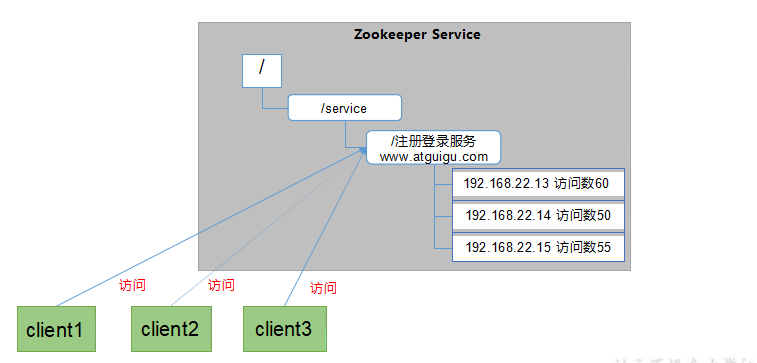
（2）监听这个ZNode可获取它的实时状态变化。



5、服务器动态上下线

客户端能实时洞察到服务器上下线的变化

6、软负载均衡



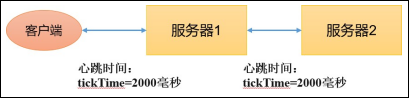
在Zookeeper中记录每台服务器的访问数，让访问数最少的服务器去处理最新的客户端请求。

# Zookeeper安装

## Zookeeper配置文件

|  |
| --- |
| tickTime=2000  initLimit=10  syncLimit=5  dataDir=/data/zookeeper  clientPort=2181 |

1）tickTime = 2000：通信心跳时间，Zookeeper服务器与客户端心跳时间，单位毫秒



2）initLimit = 10：LF初始通信时限

|  |
| --- |
|  |
|  |  |
|  |  |

Leader和Follower初始连接时能容忍的最多心跳数（tickTime的数量）

3）syncLimit = 5：LF同步通信时限



Leader和Follower之间通信时间如果超过syncLimit \* tickTime，Leader认为Follwer死掉，从服务器列表中删除Follwer。

4）dataDir：保存Zookeeper中的数据

注意：默认的tmp目录，容易被Linux系统定期删除，所以一般不用默认的tmp目录。

5）clientPort = 2181：客户端连接端口，通常不做修改。

## Zookeeper的安装

### 安装jdk

1、在/usr/local 创建java目录，将jdk解压。

2、在/etc/profile文件设置环境变量

|  |
| --- |
| export JAVA\_HOME=/usr/local/java/jdk1.8.0\_191  export JRE\_HOME=${JAVA\_HOME}/jre  export PATH=${PATH}:${JAVA\_HOME}/bin:${JRE\_HOME}/bin  export CLASSPATH=.:${JAVA\_HOME}/lib:${JRE\_HOME}/lib |

3、刷新配置文件

Source /etc/profile

4、测试

java或java -version

### 下载zookeeper

<https://mirrors.tuna.tsinghua.edu.cn/apache/zookeeper/>

### 上传解压zk

解压文件：tar -zxvf zookeeper-3.4.12

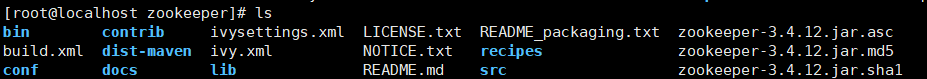
### 设置环境变量

在/etc/profile文件设置环境变量

|  |
| --- |
| export ZOOKEEPER\_HOME=/usr/local/zookeeper/apache-zookeeper-3.5.9-bin  export PATH=$PATH:$ZOOKEEPER\_HOME/bin |

source /etc/profile

### 文件目录介绍



bin:可执行文件

lib：依赖

conf：配置文件

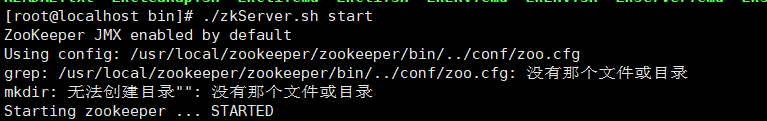
### zookeeper运行

命令： zkServer.sh start|stop|status|restart

进入bin目录：cd bin



执行文件: ./ zkServer.sh



报错原因：没有配置文件，进入conf文件夹中，将zoo\_sample.cfg修改为zoo.cfg



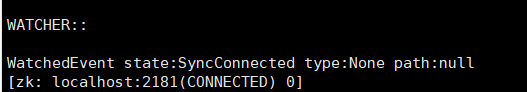
mv zoo\_sample.cfg zoo.cfg

运行zookeeper服务器：./zkServer.sh start

查看是否启动成功：jps



运行zookeeper 客户端：./zkCli.sh



默认端口：2181

### 查看日志

查看启动日志: /usr/local/zookeeper/apache-zookeeper-3.5.9-bin/logs

查看命令日志:

## 集群安装

## Zookeeper客户端

### 连接

连接客户端:bin/zkCli.sh-server ip:2181

Help:查看命令

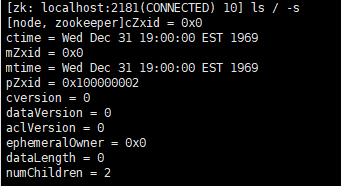
|  |  |
| --- | --- |
| 命令基本语法 | 功能描述 |
| help | 显示所有操作命令 |
| ls path | 使用 ls 命令来查看当前 znode 的子节点 [可监听]  -w 监听子节点变化  -s 附加次级信息 |

|  |  |
| --- | --- |
| create | 普通创建  -s 含有序列  -e 临时（重启或者超时消失） |
| get path | 获得节点的值 [可监听]  -w 监听节点内容变化  -s 附加次级信息 |
| set | 设置节点的具体值 |
| stat | 查看节点状态 |
| delete | 删除节点 |
| deleteall | 递归删除节点 |

|  |  |
| --- | --- |
| 命令 | 说明 |
| Create /节点 创建znode(父znode必须存在) | 新增节点并赋值 |
| Delete /节点 删除znode(zonde没有子节点) | 删除节点 |
| Set /节点 新的节点值 | 修改节点的值 |
| Get /节点 | 获取节点的值 |
| exists 测试node是否存在，并获取他的元数据 |  |

注意点：zookeeper中不能递归创建，

### znode 节点数据信息

l

（1）czxid：创建节点的事务 zxid

每次修改ZooKeeper 状态都会产生一个ZooKeeper 事务 ID。事务 ID 是ZooKeeper 中所有修改总的次序。每次修改都有唯一的 zxid，如果 zxid1 小于 zxid2，那么zxid1 在 zxid2 之前发生。

（2）ctime：znode 被创建的毫秒数（从 1970 年开始）

（3）mzxid：znode 最后更新的事务zxid

（4）mtime：znode 最后修改的毫秒数（从 1970 年开始）

（5）pZxid：znode 最后更新的子节点zxid

（6）cversion：znode 子节点变化号，znode 子节点修改次数

（7）dataversion：znode 数据变化号

（8）aclVersion：znode 访问控制列表的变化号

（9）ephemeralOwner：如果是临时节点，这个是 znode 拥有者的 session id。如果不是临时节点则是 0。

（10）dataLength：znode 的数据长度

（11）numChildren：znode 子节点数量

### 结点的监听

1）节点的值变化监听

（1）在 hadoop104 主机上注册监听/sanguo 节点数据变化

[zk: localhost:2181(CONNECTED) 26] get -w /sanguo

（2）在 hadoop103 主机上修改/sanguo 节点的数据

[zk: localhost:2181(CONNECTED) 1] set /sanguo "xisi"

（3）观察hadoop104 主机收到数据变化的监听

注意：在hadoop103再多次修改/sanguo的值，hadoop104上不会再收到监听。因为注册一次，只能监听一次。想再次监听，需要再次注册。

2）节点的子节点变化监听（路径变化）

（1）在 hadoop104 主机上注册监听/sanguo 节点的子节点变化

[shuguo, weiguo]

（2）在 hadoop103 主机/sanguo 节点上创建子节点

[zk: localhost:2181(CONNECTED) 2] create /sanguo/jin "simayi" Created /sanguo/jin

（3）观察hadoop104 主机收到子节点变化的监听

WATCHER::

WatchedEvent state:SyncConnected type:NodeChildrenChanged path:/sanguo

注意：节点的路径变化，也是注册一次，生效一次。想多次生效，就需要多次注册。

### 节点删除与查看

1）删除节点

[zk: localhost:2181(CONNECTED) 4] delete /sanguo/jin

2）递归删除节点

[zk: localhost:2181(CONNECTED) 15] deleteall /sanguo/shuguo

3）查看节点状态

|  |
| --- |
| zk: localhost:2181(CONNECTED) 17] stat /sanguo cZxid = 0x100000003  ctime = Wed Aug 29 00:03:23 CST 2018 mZxid = 0x100000011  mtime = Wed Aug 29 00:21:23 CST 2018 pZxid = 0x100000014  cversion = 9  dataVersion = 1  aclVersion = 0 ephemeralOwner = 0x0 dataLength = 4  numChildren = 1 |

# zookeeper集群

### zk集群示意图



### 设置环境变量

在/etc/profile文件设置环境变量

|  |
| --- |
| export ZOOKEEPER\_HOME=/usr/local/zookeeper/apache-zookeeper-3.5.9-bin  export PATH=$PATH:$ZOOKEEPER\_HOME/bin |

source /etc/profile

### 集群的规划

|  |  |  |
| --- | --- | --- |
| 机器 | ip | 端口 |
| zk1 | 192.168.205.129 | 2181 |
| zk2 | 192.168.205.130 | 2181 |
| zk3 | 192.168.205.131 | 2181 |

### 修改配置文件

1)创建数据数据目录

mkdir: mkdir /usr/local/zookeeper/data

2)Zk 启动时，会从数据目录里面读取一个叫myid的文件，取出里面的值，作为自己的机器编号。在每个data文件下建立一个myid的文件，依次给值1,2,3

3)修改配置文件

dataDir:Zk 启动时，会从数据目录里面读取一个叫myid的文件，取出里面的值，作为自己的机器编号。

clientPort:客户端的端口

集群的配置：每个机器的配置文件都相同

|  |
| --- |
| 集群里面编号为1 的机器, Zk 内部数据交互的端口，zk 用来选举的端口  server.1=ip:2887:3888  集群里面编号为2 的机器  server.1=ip:2887:3888  集群里面编号为3 的机器  server.1=ip:2887:3888 |

### 启动所有的zk

/usr/local/zookeeper/apache-zookeeper-3.5.9-bin/bin/zkServer.sh

### 查看集群状态

./zkServer.sh status



## Zookeeper选举机制

### 选举机制

当ZooKeeper集群中的一台服务器出现以下两种情况之一时，就会开始进入Leader选举：

1、服务器初始化启动。

2、务器运行期间无法和Leader保持连接。

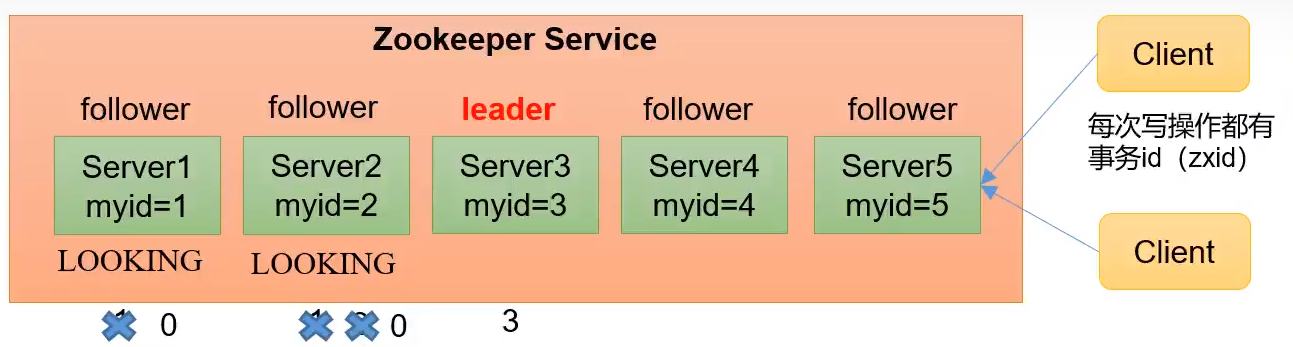
### 概念

**SID**：服务器ID。用来唯一标识一台ZooKeeper集群中的机器，每台机器不能重复，和myid一致。

**ZXID**：事务ID。ZXID是一个事务ID，用来标识一次服务器状态的变更。在某一时刻， 集群中的每台机器的ZXID值不一定完全一致，这和ZooKeeper服务器对于客户端“更 新请求”的处理逻辑有关。

Epoch：每个Leader任期的代号。没有Leader时同一轮投票过程中的逻辑时钟值是相同的。每投完一次票这个数据就会增加

### 启动选举



（1）服务器1启动，发起一次选举。服务器1投自己一票。此时服务器1票数一票，不够半数以上（3票），选举无法完成，服务器1状态保持为LOOKING；

（2）服务器2启动，再发起一次选举。服务器1和2分别投自己一票并交换选票信息：此时服务器1发现服务器2的myid比自己目前投票推举的（服务器1）大，更改选票为推举服务器2。此时服务器1票数0票，服务器2票数2票，没有半数以上结果，选举无法完成，服务器1，2状态保持LOOKING

（3）服务器3启动，发起一次选举。此时服务器1和2都会更改选票为服务器3。此次投票结果：服务器1为0票，服务器2为0票，服务器3为3票。此时服

务器3的票数已经超过半数，服务器3当选Leader。服务器1，2更改状态为FOLLOWING，服务器3更改状态为LEADING；

（4）服务器4启动，发起一次选举。此时服务器1，2，3已经不是LOOKING状态，不会更改选票信息。交换选票信息结果：服务器3为3票，服务器4为

1票。此时服务器4服从多数，更改选票信息为服务器3，并更改状态为FOLLOWING；

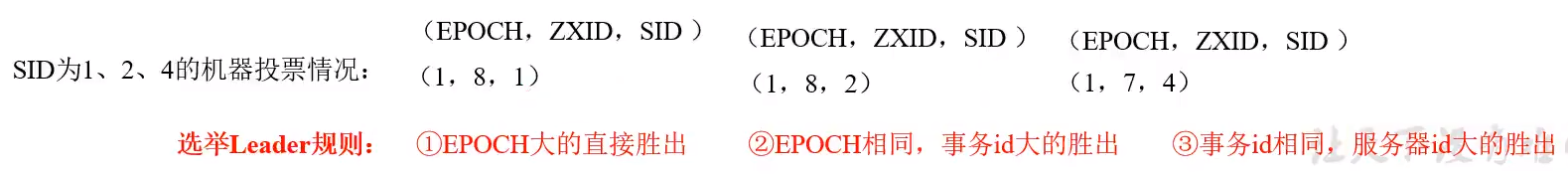
（5）服务器5启动，同4一样当小弟。

### 重新选举

而当一台机器进入Leader选举流程时，当前集群也可能会处于以下两种状态：

1、集群中本来就已经存在一个Leader。对于第一种已经存在Leader的情况，机器试图去选举Leader时，会被告知当前服务器的Leader信息，对于该机器来说，仅仅需要和Leader机器建立连接，并进行状态同步即可。

2、集群中确实不存在Leader。假设ZooKeeper由5台服务器组成，SID分别为1、2、3、4、5，ZXID分别为8、8、8、7、7，并且此时SID为3的服务器是Leader。某一时刻，3和5服务器出现故障，因此开始进行Leader选举。



# Zookeeper的特性

## 动态配置

### 引入

手动集群成员的调整

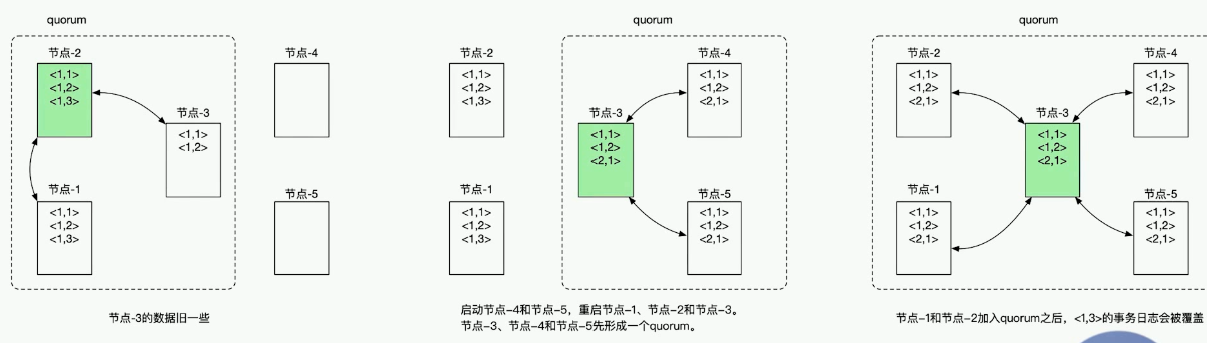
1,停止整个ZooKeeper现有集群。

2·更改配置文件zoo.cfg的server.n项。

3·启动新集群的Zookeeper节点。

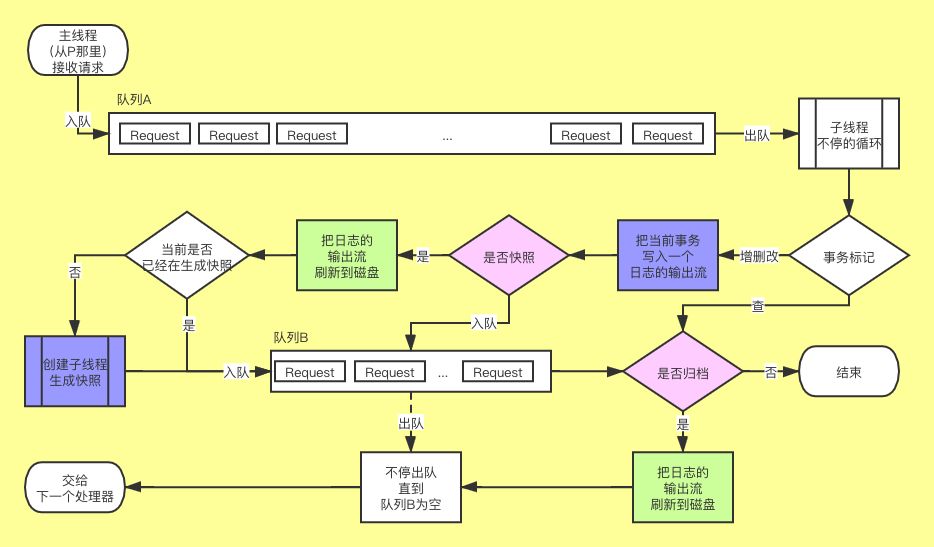
问题1 :需要停止Zookeeper服务。

问题2:可能会导致已经提交的数据写入被覆盖。

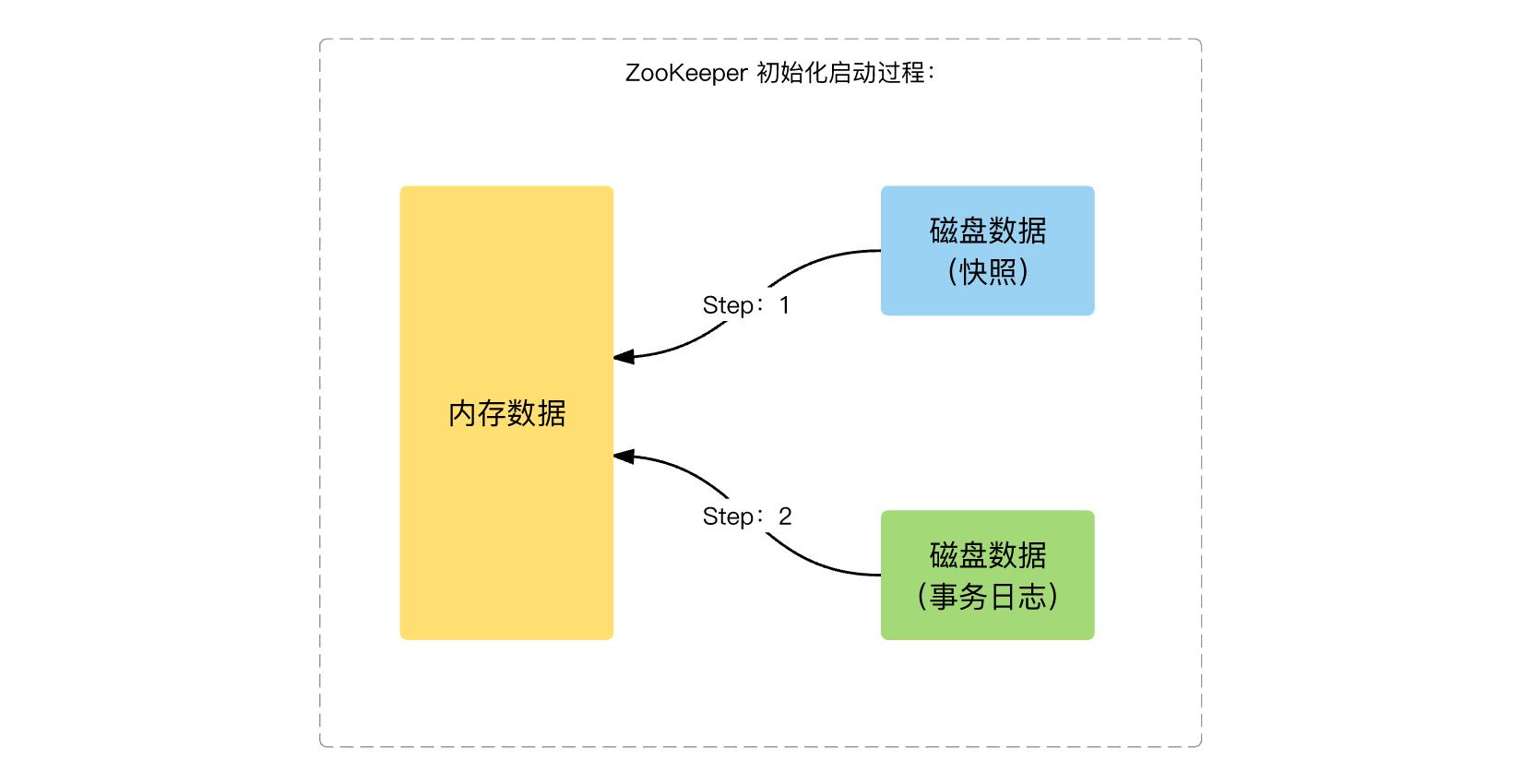


### 动态配置

## 持久化



### zookeeper数据



整体分为 3 类：

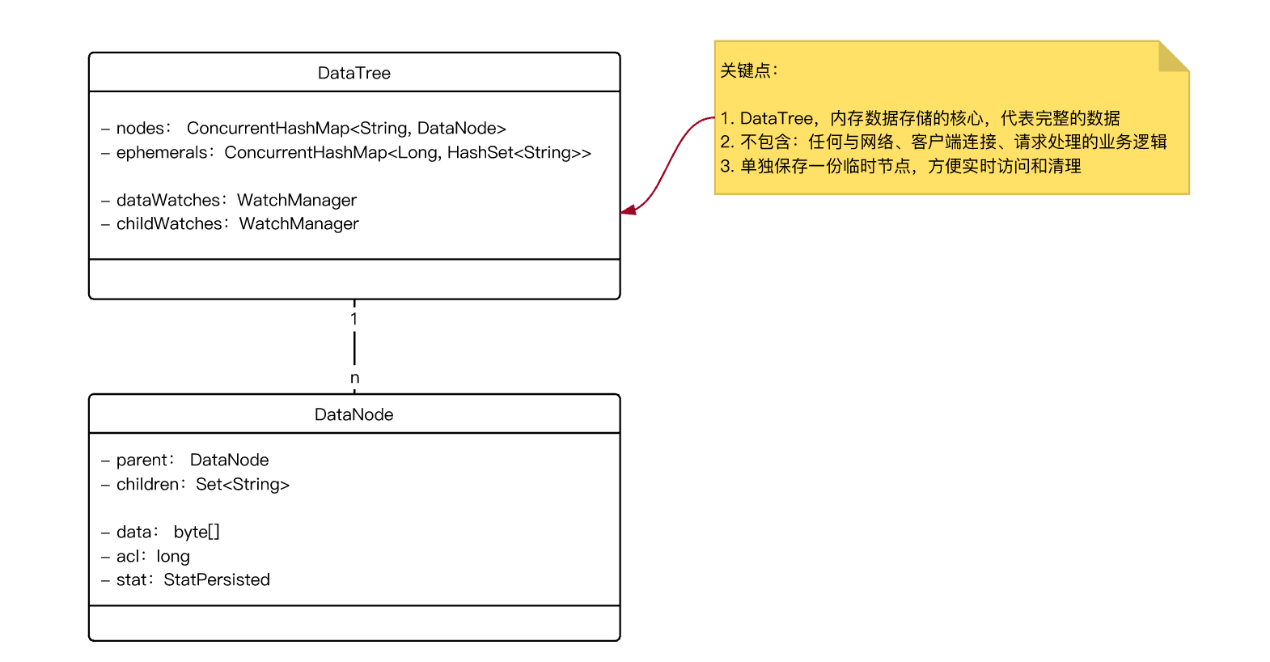
**内存数据**

**磁盘数据**

**快照**

**事务日志**

### 内存数据



ZK 的数据模型：树

树中单个节点包含的内容：节点数据、节点 ACL 信息、节点的路径

Node: 持久化结点

Key: 唯一标识 path 作为 key

value DataNode: 节点数据、节点 ACL 信息、节点的路径

Ephemerals: 存储临时节点

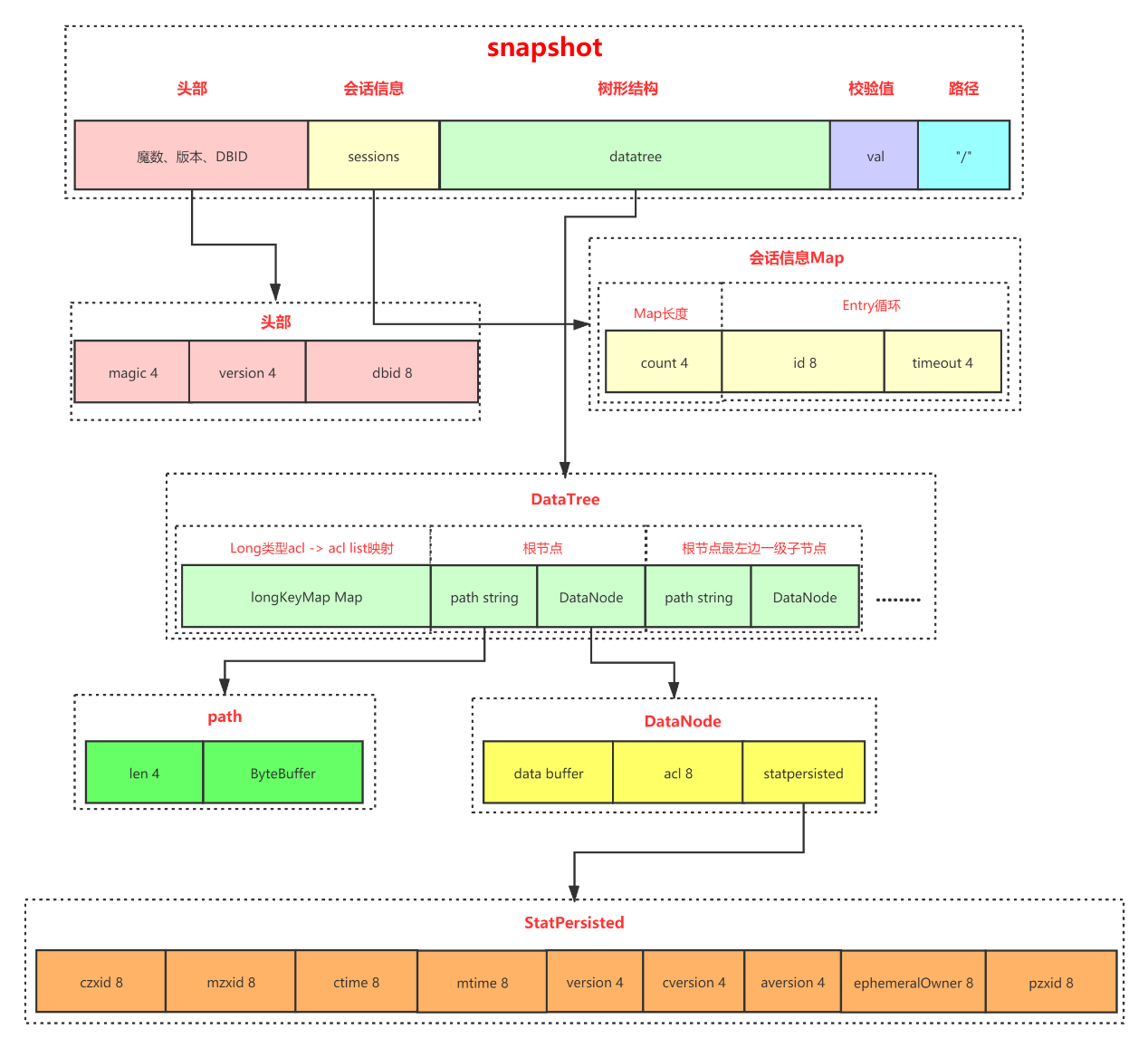
Key: 临时节点是跟 Session 绑定的，sessionId 作为 key

Value:

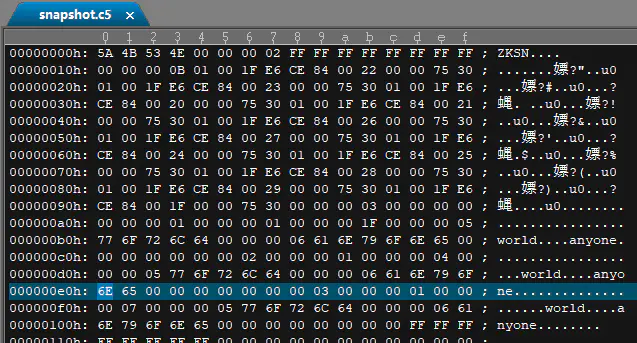
=

### 快照

文件结构



序列化内容snapshot.xx



内容解析

|  |
| --- |
| ZNode Details (count=22):  ----  /  cZxid = 0x00000000000000  ctime = Thu Jan 01 08:00:00 CST 1970  mZxid = 0x00000000000000  mtime = Thu Jan 01 08:00:00 CST 1970  pZxid = 0x0000000000007c  cversion = 10  dataVersion = 0  aclVersion = 0  ephemeralOwner = 0x00000000000000  dataLength = 0  ----  。。。  Session Details (sid, timeout, ephemeralCount):  0x1001fe6ce840022, 30000, 0  0x1001fe6ce840023, 30000, 0  0x1001fe6ce840020, 30000, 0  0x1001fe6ce840021, 30000, 0  0x1001fe6ce840026, 30000, 0  0x1001fe6ce840027, 30000, 0  0x1001fe6ce840024, 30000, 0  0x1001fe6ce840025, 30000, 0  0x1001fe6ce840028, 30000, 0  0x1001fe6ce840029, 30000, 0  0x1001fe6ce84001f, 30000, 0 |

### 事务日志(txnlog)

每个执行的事务都会写入到事务日志中,其存储位置由dataLogDir配置,当未配置dataLogDir时,使用dataDir作为存储目录,由于事务日志的写入速度较为影响Zookeeper的性能,可以将dataLogDir单独配置到一块磁盘上

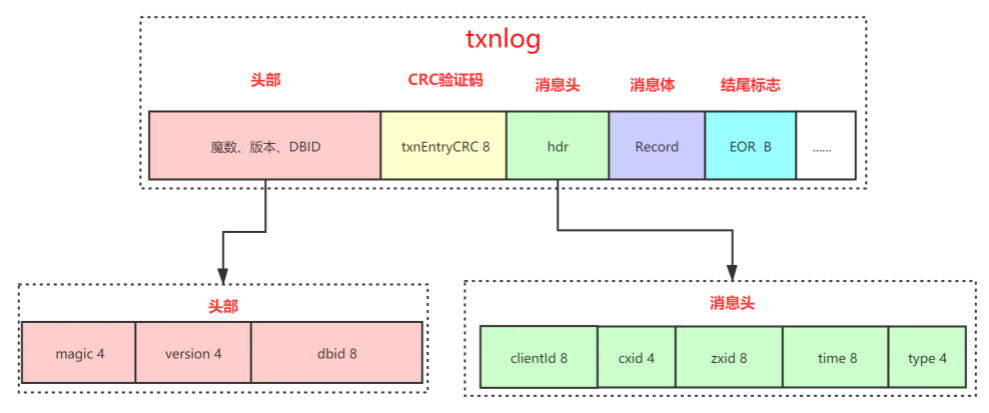
由于事务日志要不断的写入,会触发底层磁盘I/O为文件开辟新的磁盘块,为了减少分配新磁盘块对写入的影响,Zookeeper使用预分配策略,默认每次分配新文件或扩容时,一次分配64MB

扩容事务日志文件时机:初始化事务日志文件时为其分配64MB,当写入事务日志的过程中,发现剩余可写入空间小于4KB时,进行扩容,依然是为事务日志文件增加64MB

生成新事务日志文件时机:即使当前事务日志文件可写空间较少,也只会进行扩容,不会生成新的事务日志文件.在经过snapCount次事务后,会生成快照文件,但同时将当前事务日志的输出流置null,这样下次写事务日志时自动创建新的事务日志文件

为了便于快速根据zxid找到存储该zxid对应事务的事务日志文件,事务日志文件的命名是有意义的,事务日志文件的命名为log.{zxid},后缀是该日志文件存储的第一个事务的zxid

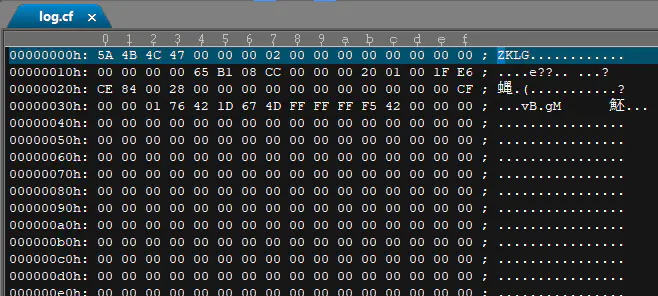
内容结构:



每一次增量日志包含4部分：CRC验证码+消息头+消息体+结尾标志

Record：不同请求命令对应不同的Record；

**序列化内容log.cf**



内容解析;

|  |
| --- |
| ZooKeeper Transactional Log File with dbid 0 txnlog format version 2  20-12-6 下午12时39分36秒 session 0x100142dc1660000 cxid 0x0 zxid 0x1e createSession 30000  20-12-6 下午12时39分52秒 session 0x100141757900001 cxid 0x0 zxid 0x1f closeSession null  20-12-6 下午12时39分59秒 session 0x100142dc1660000 cxid 0x2 zxid 0x20 create '/zk/test5,#353535,v{s{31,s{'world,'anyone}}},F,6  20-12-6 下午12时53分04秒 session 0x100142dc1660000 cxid 0x0 zxid 0x21 closeSession null  20-12-6 下午12时53分04秒 session 0x100142dc1660001 cxid 0x0 zxid 0x22 createSession 30000  20-12-6 下午12时53分36秒 session 0x100142dc166001d cxid 0x0 zxid 0x5f closeSession null  20-12-6 下午12时53分36秒 session 0x100142dc1660005 cxid 0x0 zxid 0x60 closeSession null  20-12-6 下午12时54分56秒 session 0x100142dc1660020 cxid 0x4 zxid 0x61 setACL '/zk/test1,v{s{4,s{'world,'anyone}}},1  20-12-6 下午12时55分39秒 session 0x100142dc1660020 cxid 0x6 zxid 0x62 create '/zk/test7,#373737,v{s{4,s{'world,'anyone}}},F,7  20-12-6 下午02时33分27秒 session 0x100142dc1660020 cxid 0x0 zxid 0x63 closeSession null  EOF reached after 70 txns. |

## java 操作zookeeper

### maven项目添加依赖()

java+zookeeper=zkclient

依赖

|  |
| --- |
| <!-- https://mvnrepository.com/artifact/com.101tec/zkclient -->  <dependency>  <groupId>com.101tec</groupId>  <artifactId>zkclient</artifactId>  <version>0.10</version>  </dependency> |

zookeeper.jar：zookeeper原生的操作

zlclient:原生操作的基础上进行的封装。

### 测试连接、

|  |
| --- |
| **public** **class** app {  **private** **static** ZkClient *client* = **null**;  **static** {  *client* = **new** ZkClient("192.168.10.136:2181");  }  //测试连接  @Test  **public** **void** test1() {  //获取根路径下的所有节点  List<String> children = *client*.getChildren("/");  System.***out***.println(children);  } |

### 增加节点

|  |
| --- |
| //添加一个节点  @Test  **public** **void** test2() {  //根路径下添加节点  String path = "/testZK";  *client*.create(path, **null**, CreateMode.***PERSISTENT***);  **boolean** exists = *client*.exists(path);  **if**(exists) {  System.***out***.println("create ok");  }**else** {  System.***out***.println("fail");  }  } |

### 删除节点

|  |
| --- |
| //删除一个节点,不能递归删除  @Test  **public** **void** test3() {  **boolean** delete = *client*.delete("/testZK");  **if**(delete) {  System.***out***.println("delete ok");  }**else** {  System.***out***.println("fail");  }  } |

### 修改节点数据

|  |
| --- |
| //修改节点的数据  @Test  **public** **void** test4() {  //创建节点并设置值  *client*.createPersistent("/test/test1","value");  //获取节点的值  Object olDdata = *client*.readData("/test/test1");  System.***out***.println(olDdata);  //修改节点的值  *client*.writeData("/test/test1", "value1");  Object newData = *client*.readData("/test/test1");  System.***out***.println(newData);  } |

### 递归创建节点

|  |
| --- |
| //递归创建节点  @Test  **public** **void** test5() {  //1、递归创建持久节点  //path:节点路径  //createParent:是否创建自动创建父节点  *client*.createPersistent("/node/node2/node2",**true**);  //注:不能递归创建临时节点,会报错。  *client*.close();  } |

### 订阅一个节点

如果节点的变化，会通知

子节点的变化

节点的删除

节点的修改

子节点的添加和删除

|  |
| --- |
| //监听子节点的变化  //字节点的添加和删除  **public** **static** **void** main(String[] args) {  String path = "/node/node1";  *client*.subscribeChildChanges(path, **new** IZkChildListener() {  //parentPath:节点的父路径  //currentChilds:  // 如果是删除，返回null  // 如果是添加，返回当前子节点  @Override  **public** **void** handleChildChange(String parentPath,  List<String> currentChilds) **throws** Exception {  String name = Thread.*currentThread*().getName();  System.***out***.println("监听器的线程为"+name);  //监听器的线程为ZkClient-EventThread-11-192.168.10.136:2181  //删除parentPath:node/node1,currentChilds:[]  //添加parentPath:node/node1,currentChilds:[node3]  System.***out***.println(parentPath);  System.***out***.println(currentChilds);  }  } );  //阻塞主线程  **try** {  System.***in***.read();  } **catch** (IOException e) {  e.printStackTrace();  }  } |

子节点的变化

|  |
| --- |
| //监听值的变化  **public** **static** **void** testDataChanges() {  String path = "/node/node1/node3";  //监听节点的值是否发生改变  *client*.subscribeDataChanges(path, **new** IZkDataListener() {  //监听是节点的值是否删除  @Override  **public** **void** handleDataDeleted(String dataPath) **throws** Exception {  System.***out***.println(Thread.*currentThread*().getName());  System.***out***.println("数据删除了"+dataPath);  }    //监听节点的值是否改变  /\*\*  \* 进来了，但是没有显示出来，因为出异常了  \* Object data 修改后的值  \* Object  \* zkCli 设置的是String 值  \* java 设置的是Object值  \*/  @Override  **public** **void** handleDataChange(String dataPath, Object data) **throws** Exception {  System.***out***.println(Thread.*currentThread*().getName());  System.***out***.println("数据改变了"+dataPath);  }  });  //阻塞主线程  **try** {  System.***in***.read();  } **catch** (IOException e) {  e.printStackTrace();  }  }  //修该值  @Test  **public** **static** **void** updateNode() {  *client*.writeData("/node/node1/node3", "xx111");  } |

# Zookeeper源码

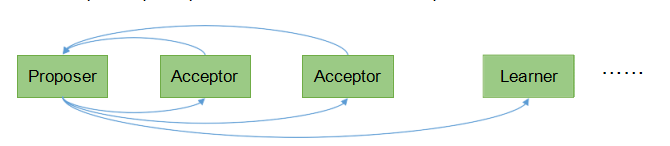
## Paxos算法

### Paxos算法

Paxos算法：一种基于消息传递且具有高度容错特性的一致性算法。

Paxos算法解决的问题：就是如何快速正确的在一个分布式系统中对某个数据值达成一致，并且保证不论发生任何异常，都不会破坏整个系统的一致性。

### 算法描述



在一个Paxos系统中，首先将所有节点划分为Proposer（提议者），Acceptor（接受者），和Learner（学习者）。（注意：每个节点都可以身兼数职）。

一个完整的Paxos算法流程分为三个阶段：

Prepare准备阶段

Proposer向多个Acceptor发出Propose请求Promise（承诺）

Acceptor针对收到的Propose请求进行Promise（承诺）

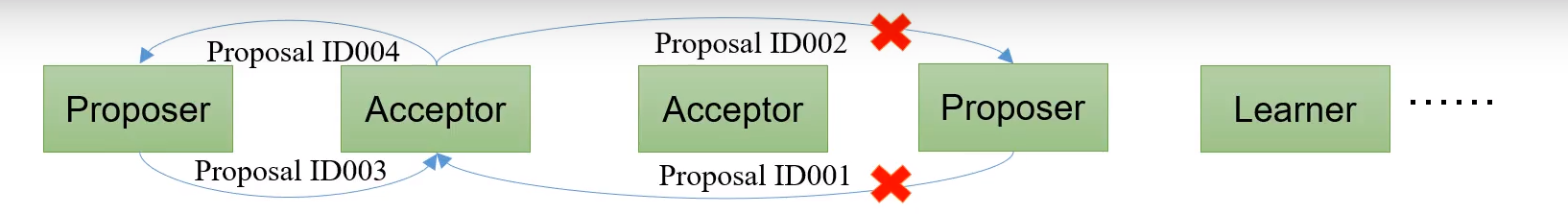
Accept接受阶段

Proposer收到多数Acceptor承诺的Promise后，向Acceptor发出Propose请求

Acceptor针对收到的Propose请求进行Accept处理

Learn学习阶段：Proposer将形成的决议发送给所有Learners

### 详细过程



（1）Prepare: Proposer生成全局唯一且递增的Proposal ID，向所有Acceptor发送Propose请求，这里无需携带提案内容，只携带Proposal ID即可。

（2）Promise: Acceptor收到Propose请求后，做出“两个承诺，一个应答”。

不再接受Proposal ID小于等于（注意：这里是<= ）当前请求的Propose请求。

不再接受Proposal ID小于（注意：这里是< ）当前请求的Accept请求。

不违背以前做出的承诺下，回复已经Accept过的提案中Proposal ID最大的那个提案的Value和Proposal ID，没有则返回空值。

（3）Propose: Proposer收到多数Acceptor的Promise应答后，从应答中选择Proposal ID最大的提案的Value，作为本次要发起的提案。如果所有应答的提案Value均为空值，则可以自己随意决定提案Value。然后携带当前Proposal ID，向所有Acceptor发送Propose请求。

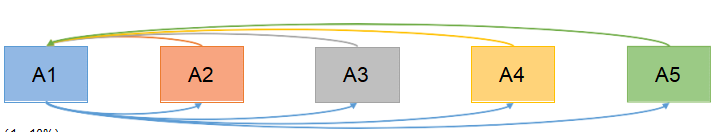
（4）Accept: Acceptor收到Propose请求后，在不违背自己之前做出的承诺下，接受并持久化当前Proposal ID和提案Value。

（5）Learn: Proposer收到多数Acceptor的Accept后，决议形成，将形成的决议发送给所有Learner。

### 例如

情况1:

有A1, A2, A3, A4, A5 5位议员，就税率问题进行决议。



A1发起1号Proposal的Propose，等待Promise承诺；

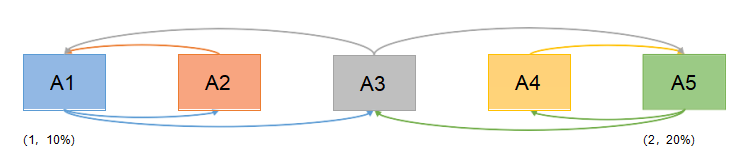
A2-A5回应Promise；

A1在收到两份回复时就会发起税率10%的Proposal；(超过半数)

A2-A5回应Accept；

通过Proposal，税率10%。

情况2: 现在我们假设在A1提出提案的同时, A5决定将税率定为20%



A1，A5同时发起Propose（序号分别为1，2）

A2承诺A1，A4承诺A5，A3行为成为关键

情况1：A3先收到A1消息，承诺A1。

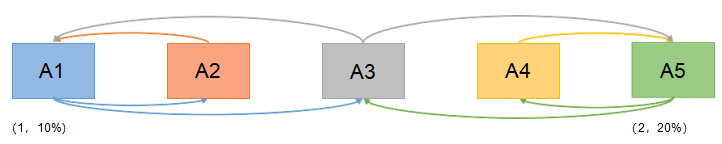
A1发起Proposal（1，10%），A2，A3接受。

之后A3又收到A5消息，回复A1：（1，10%），并承诺A5。

A5发起Proposal（2，20%），A3，A4接受。之后A1，A5同时广播决议。

情况3：

现在我们假设在A1提出提案的同时, A5决定将税率定为20%



A1，A5同时发起Propose（序号分别为1，2）

A2承诺A1，A4承诺A5，A3行为成为关键

情况2：A3先收到A1消息，承诺A1。之后立刻收到A5消息，承诺A5。

A1发起Proposal（1，10%），无足够响应，A1重新Propose （序号3），A3再次承诺A1。

A5发起Proposal（2，20%），无足够相应。A5重新Propose （序号4），A3再次承诺A5。

造成这种情况的原因是系统中有一个以上的 Proposer，多个 Proposers 相互争夺Acceptor，造成迟迟无法达成一致的情况。针对这种情况，一种改进的 Paxos 算法被提出：从系统中选出一个节点作为 Leader，只有 Leader 能够发起提案。这样，一次 Paxos 流程中只有一个Proposer，不会出现活锁的情况，此时只会出现例子中第一种情况。

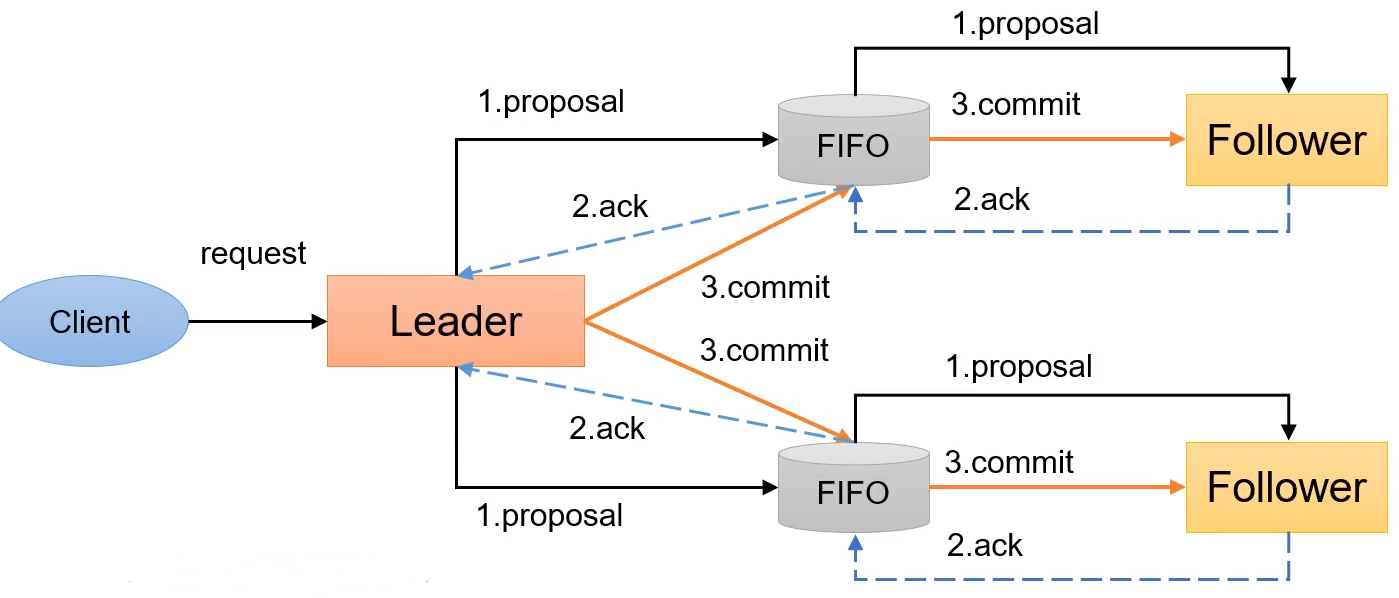
## Zab协议

### 什么是 ZAB 算法

Zab 借鉴了 Paxos 算法，是特别为 Zookeeper 设计的支持崩溃恢复的原子广播协议。基于该协议，Zookeeper 设计为只有一台客户端（Leader）负责处理外部的写事务请求，然后Leader 客户端将数据同步到其他 Follower 节点。即 Zookeeper 只有一个 Leader 可以发起提案。

Zab 协议包括两种基本的模式：消息广播、崩溃恢复。

### 消息广播



（1）客户端发起一个写操作请求。

（2）Leader服务器将客户端的请求转化为事务Proposal 提案，同时为每个Proposal 分配一个全局的ID，即zxid。

（3）Leader服务器为每个Follower服务器分配一个单独的队列，然后将需要广播的 Proposal依次放到队列中去，并且根据FIFO策略进行消息发送。

（4）Follower接收到Proposal后，会首先将其以事务日志的方式写入本地磁盘中，写入成功后向Leader反馈一个Ack响应消息。

（5）Leader接收到超过半数以上Follower的Ack响应消息后，即认为消息发送成功，可以发送commit消息。

（6）Leader向所有Follower广播commit消息，同时自身也会完成事务提交。Follower 接收到commit消息后，会将上一条事务提交。

（7）Zookeeper采用Zab协议的核心，就是只要有一台服务器提交了Proposal，就要确保所有的服务器最终都能正确提交Proposal。

问题：

ZAB协议针对事务请求的处理过程类似于一个两阶段提交过程

（1）广播事务阶段

（2）广播提交操作

这两阶段提交模型如下，有可能因为Leader宕机带来数据不一致，比如

（1）Leader 发 起 一 个 事 务Proposal1 后就宕机， Follower 都没有Proposal1

（2） Leader收到半数ACK宕机，没来得及向Follower发送Commit

### 崩溃恢复



1、异常情况:

（1）假设一个事务在Leader提出之后，Leader挂了。

（2）一个事务在Leader上提交了，并且过半的Follower都响应Ack了，但是Leader在Commit消息发出之前挂了。

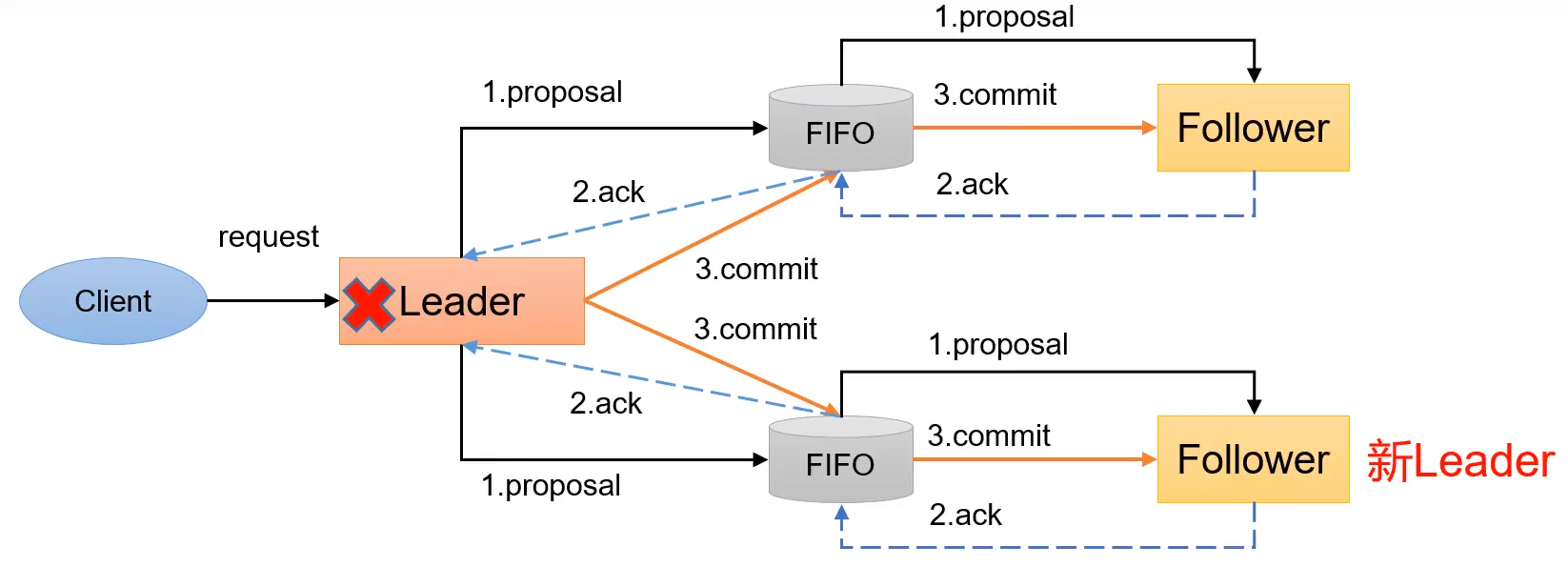
2）Zab协议崩溃恢复要求满足以下两个要求：

（1）确保已经被Leader提交的提案Proposal，必须最终被所有的Follower服务器提交。 （已经产生的提案，Follower必须执行）

（2）确保丢弃已经被Leader提出的，但是没有被提交的Proposal。（丢弃胎死腹中的提案）

### 崩溃恢复-leader选举

崩溃恢复主要包括两部分：Leader选举和数据恢复。



Leader选举：根据上述要求，Zab协议需要保证选举出来的Leader需要满足以下条件：

（1）新选举出来的Leader不能包含未提交的Proposal。即新Leader必须都是已经提交了Proposal的Follower服务器节点。

（2）新选举的Leader节点中含有最大的zxid。这样做的好处是可以避免Leader服务器检查Proposal的提交和丢弃工作。

数据同步:

（1）完成Leader选举后，在正式开始工作之前（接收事务请求，然后提出新的Proposal），Leader服务器会首先确认事务日志中的所有的Proposal 是否已经被集群中过半的服务器Commit。

（2）Leader服务器需要确保所有的Follower服务器能够接收到每一条事务的Proposal，并且能将所有已经提交的事务Proposal 应用到内存数据中。等到Follower将所有尚未同步的事务Proposal都从Leader服务器上同步过，并且应用到内存数据中以后，Leader才会把该Follower加入到真正可用的Follower列表中。

## Cap理论

### Cap

CAP理论告诉我们，一个分布式系统不可能同时满足以下三种

一致性（C:Consistency）

可用性（A:Available）

分区容错性（P:Partition Tolerance）

这三个基本需求，最多只能同时满足其中的两项，因为P是必须的，因此往往选择就在CP或者AP中。

1）） 一致性（ C:Consistency）

在分布式环境中，一致性是指数据在多个副本之间是否能够保持数据一致的特性。在一致性的需求下，当一个系统在数据一致的状态下执行更新操作后，应该保证系统的数据仍然处于一致的状态。

2）可用性（A:Available）

可用性是指系统提供的服务必须一直处于可用的状态，对于用户的每一个操作请求总是能够在有限的时间内返回结果。

3）分区容错性（P:Partition Tolerance）

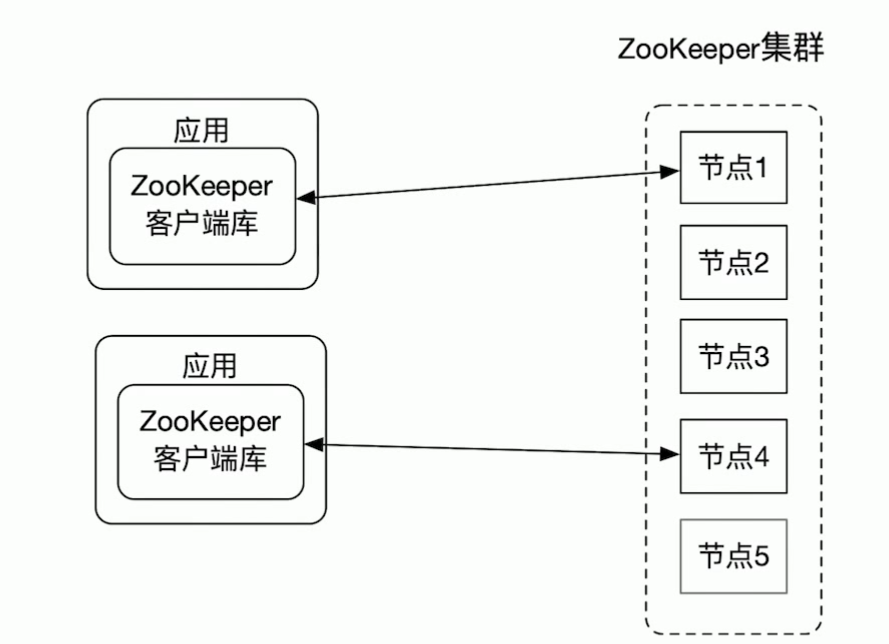
分布式系统在遇到任

ZooKeeper保证的是CP

（1）ZooKeeper不能保证每次服务请求的可用性。（注：在极端环境下，ZooKeeper可能会丢弃一些请求，消费者程序需要重新请求才能获得结果）。所以说，ZooKeeper不能保证服务可用性。

（2）进行Leader选举时集群都是不可用。

## Zookeeper架构



应用使用Zookeeper客户端库使用Zookeeper服务。ZooKeeper客户端负责和ZooKeeper集群的交互。ZooKeeper集群可以有两种模式: standalone 模式和quorum模式。处于standalone模式的ZooKeeper集群还有一个独立运行的Zookeeper节点。处于quorum模式的Zookeeper集群包含多个Zookeeper节点。

### Quorum

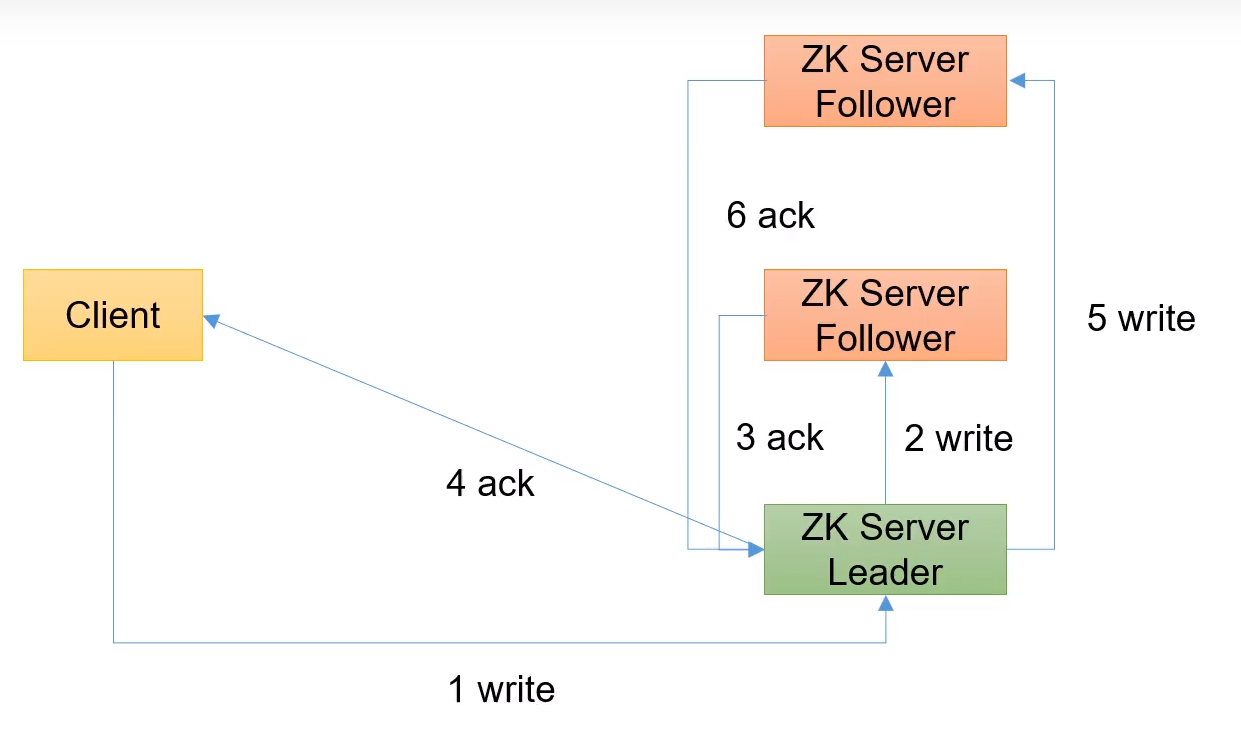
处于Quorum模式的Zookeeper集群包含多个Zookeeper节点。下图的Zookeeper集群有3个节点,其中节点1是leader节点,节点2和节点3是follower节点。leader节点可以处理读写请求, follower只可以处理读请求。follower在接到写请求时会把写请求转发给leader来处理。

### 数据一致性

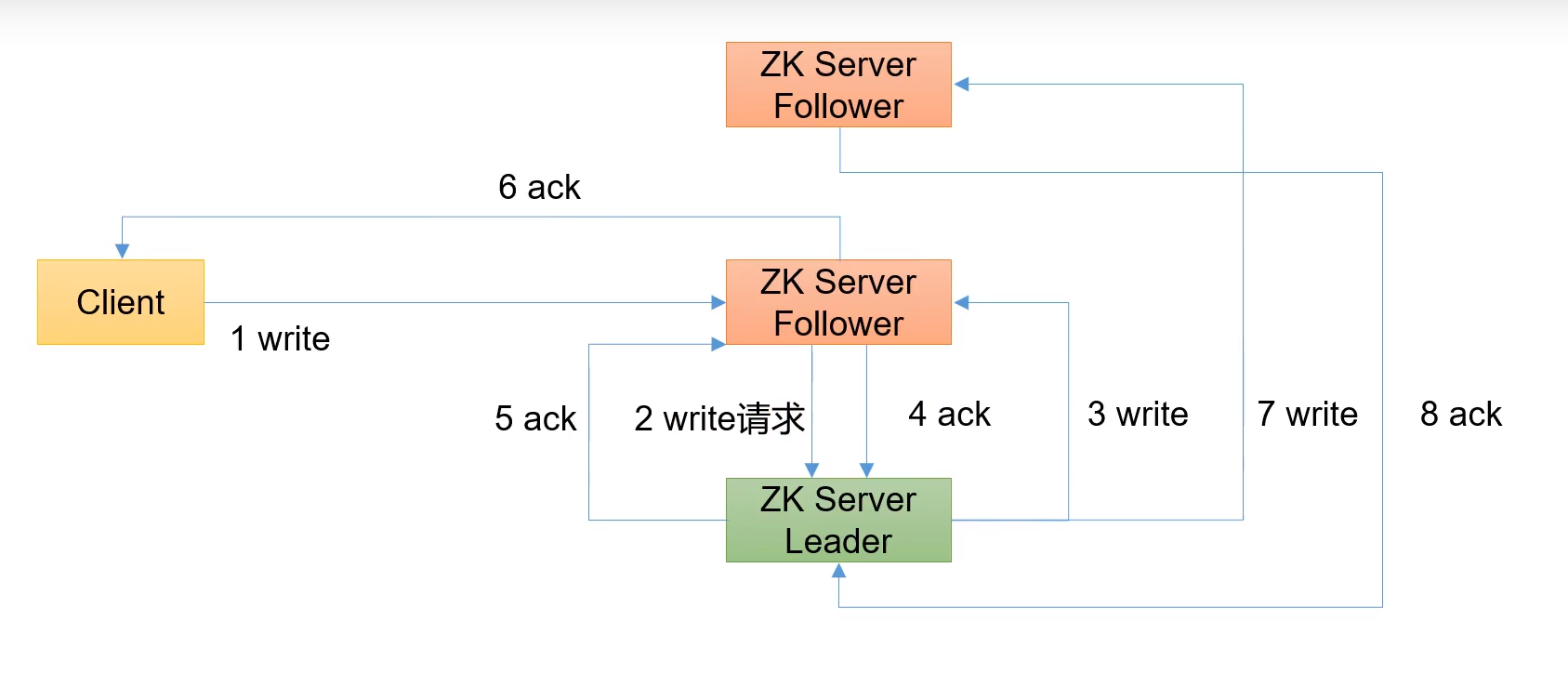
全局可线性化( Linearizable )写入:先到达leader的写请求会被先处理, leader决定写请求的执行顺序。客户端FIFO顺序:来自给定客户端的请求按照发送顺序执行。

## 写数据原理

### 访问leader



### 访问的从结点



1、客户端发送写请求给结点1

2、结点1会转发给结点2。、

3、结点2发送propose给所有的结点。

4、结点2收到 超过半数结点的accept之后，给所有的结点发送commit信息。

5、结点1发送给客户端确认消息。

## 持久化

### 持久化

Leader 和 Follower 中的数据会在内存和磁盘中各保存一份。所以需要将内存中的数据持久化到磁盘中。

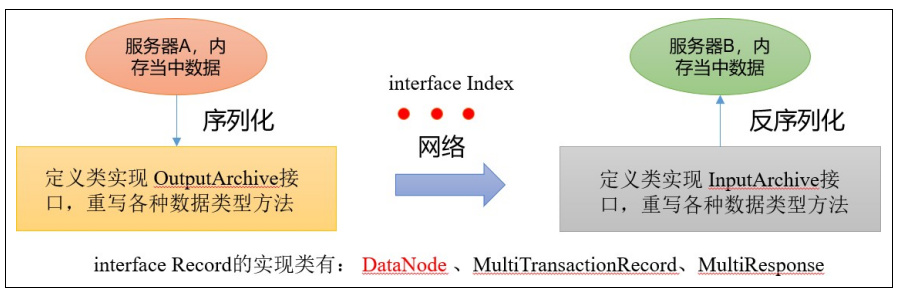
在 org.apache.zookeeper.server.persistence 包下的相关类都是序列化相关的代码。

|  |
| --- |
|  |
|  |  |



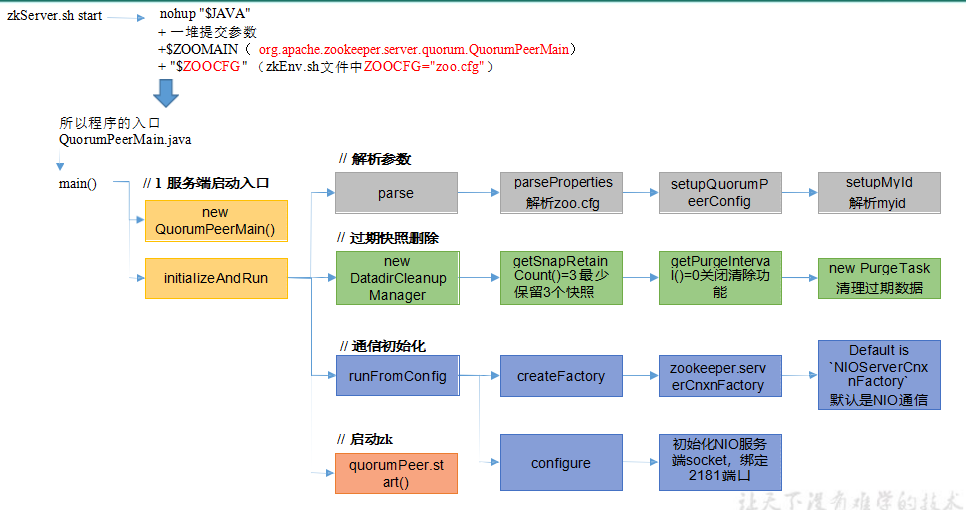
### 序列化

zookeeper-jute 代码是关于Zookeeper 序列化相关源码



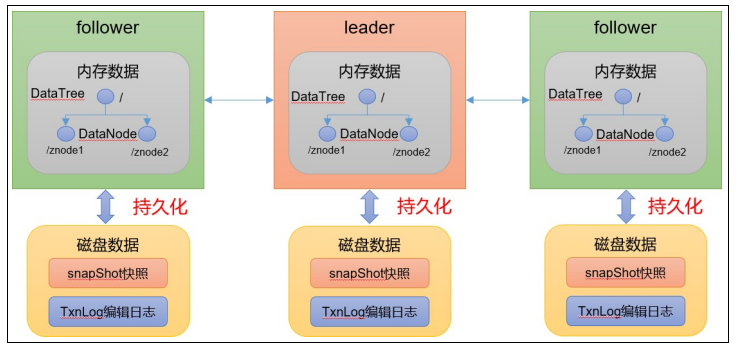
## 服务员源码

### 初始化



## ZK 服务端加载数据源码解析

### 数据持久化



（1）zk 中的数据模型，是一棵树，DataTree，每个节点，叫做DataNode

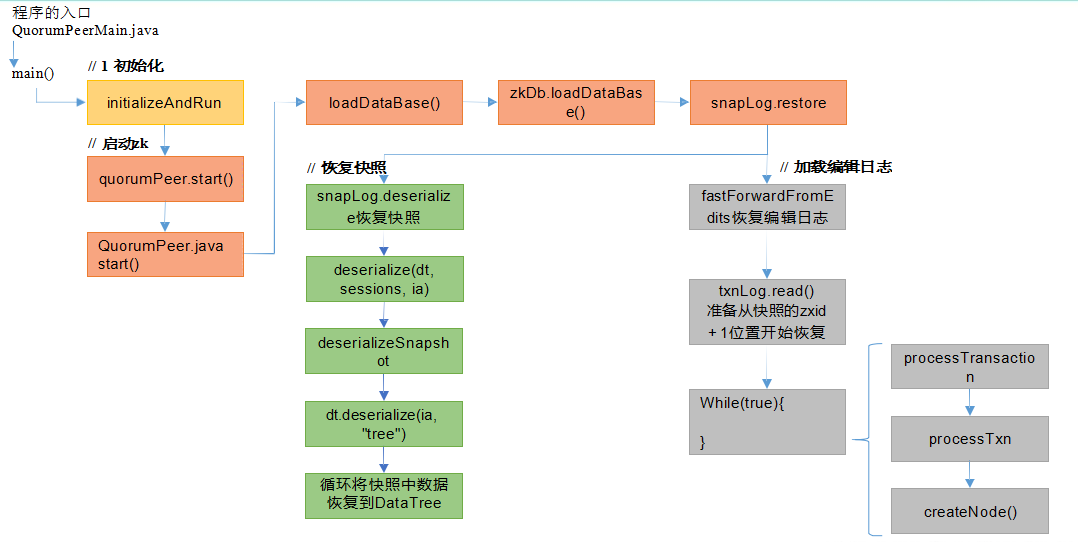
（2）zk 集群中的DataTree 时刻保持状态同步

（3）Zookeeper 集群中每个 zk 节点中，数据在内存和磁盘中都有一份完整的数据。

内存数据：DataTree

磁盘数据：快照文件 + 编辑日志

### 流程



启动之后:恢复快照数据和加载编辑日志，两者加起来为全部的数据

**恢复快照数据datatree**

**1、反序列化加载快照文件。**

**2、数据的解析，将持久化数据恢复成内存datatree。**

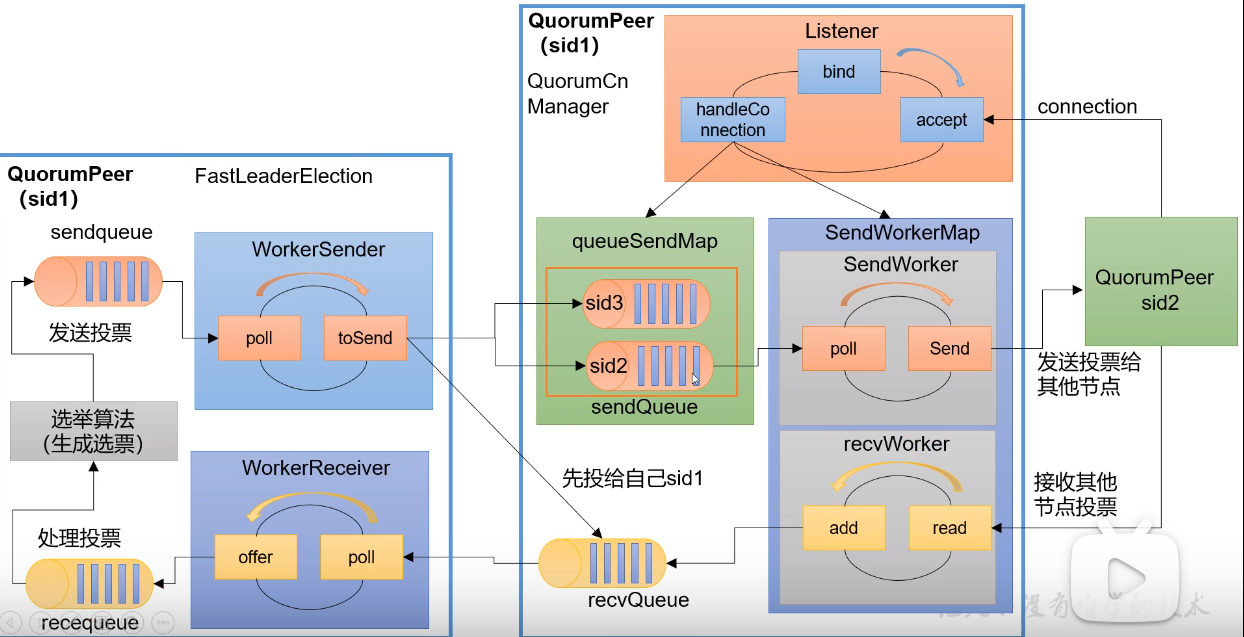
**加载编辑日志datatree:**

**1、在此之前，已经从快照文件中恢复了大部分数据，接下来只需从快照的 zxid + 1位置开始恢复。**

**2、直到回复到最达的zxid事务的最大id。**

## Learder选举

### Leader选举(looking状态)



1、生成选票,有serverid zixd事务id， epoch:leader任期的代号。将该信息放linkendblockqueue阻塞队列中。

2、生成一个worker(新的线程),从队列中获取，发送。如果当前的myid和发送的myid一致，发送给自己。否则发送给其他结点。将该结点信息发送的发送队列(ArrayBlockingQueue)

3、sendWorder将信息发送给其他的机器，然后receiveworker接收返回的数据。

4、workerreceive会接收revqueue的数据，然后写入的recequeue队列中。

5、然后选举算法判断是是否超过了半数。

### Follower和leader的状态同步

当选举结束后，每个节点都需要根据自己的角色更新自己的状态。选举出的 Leader 更新自己状态为Leader，其他节点更新自己状态为 Follower。

Leader 更新状态入口：leader.lead()

Follower 更新状态入口：follower.followerLeader()

（1）follower 必须要让 leader 知道自己的状态：epoch、zxid、sid 必须要找出谁是leader；

发起请求连接 leader；

发送自己的信息给leader；

leader 接收到信息，必须要返回对应的信息给 follower。

（2）当leader 得知follower 的状态了，就确定需要做何种方式的数据同步DIFF、TRUNC、SNAP

（3）执行数据同步

（4）当 leader 接收到超过半数 follower 的 ack 之后，进入正常工作状态，集群启动完成了

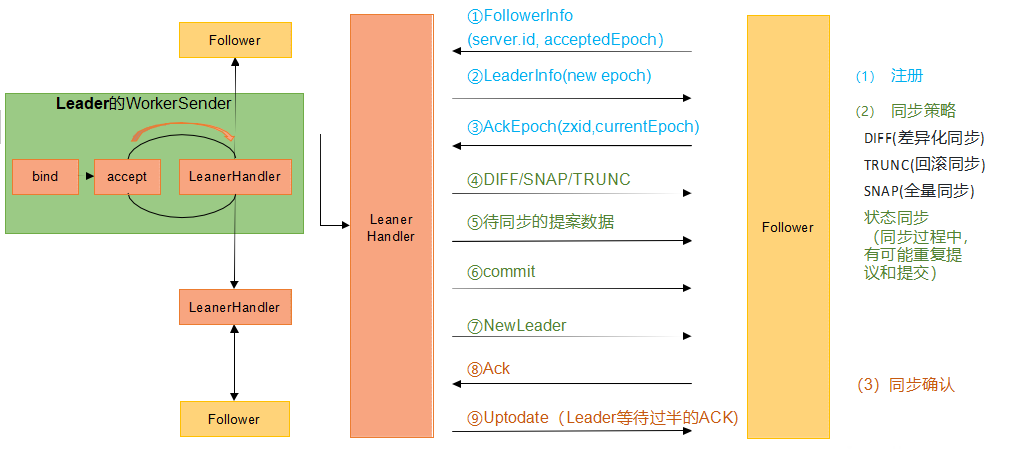
最终总结同步的方式：

（1）DIFF 咱两一样，不需要做什么

（2）TRUNC follower 的 zxid 比 leader 的 zxid 大，所以 Follower 要回滚

（3）COMMIT leader 的zxid 比 follower 的 zxid 大，发送 Proposal 给 foloower 提交执行

（4）如果 follower 并没有任何数据，直接使用 SNAP 的方式来执行数据同步（直接把数据全部序列到follower）



1、follower获取leader的机器，与leader建立链接请求。follower发送servieid epochid、和zxid发送给leader。

2、lerader发送新的epochid给follower。Follower ack该消息。

3、leader发送给follower同步方案

Diff 差异化同步

Trunc:回滚同步

Snap:轻量同步

4、follow返回给lerader同步数据方案。

5、leader发给follower待同步的数据。

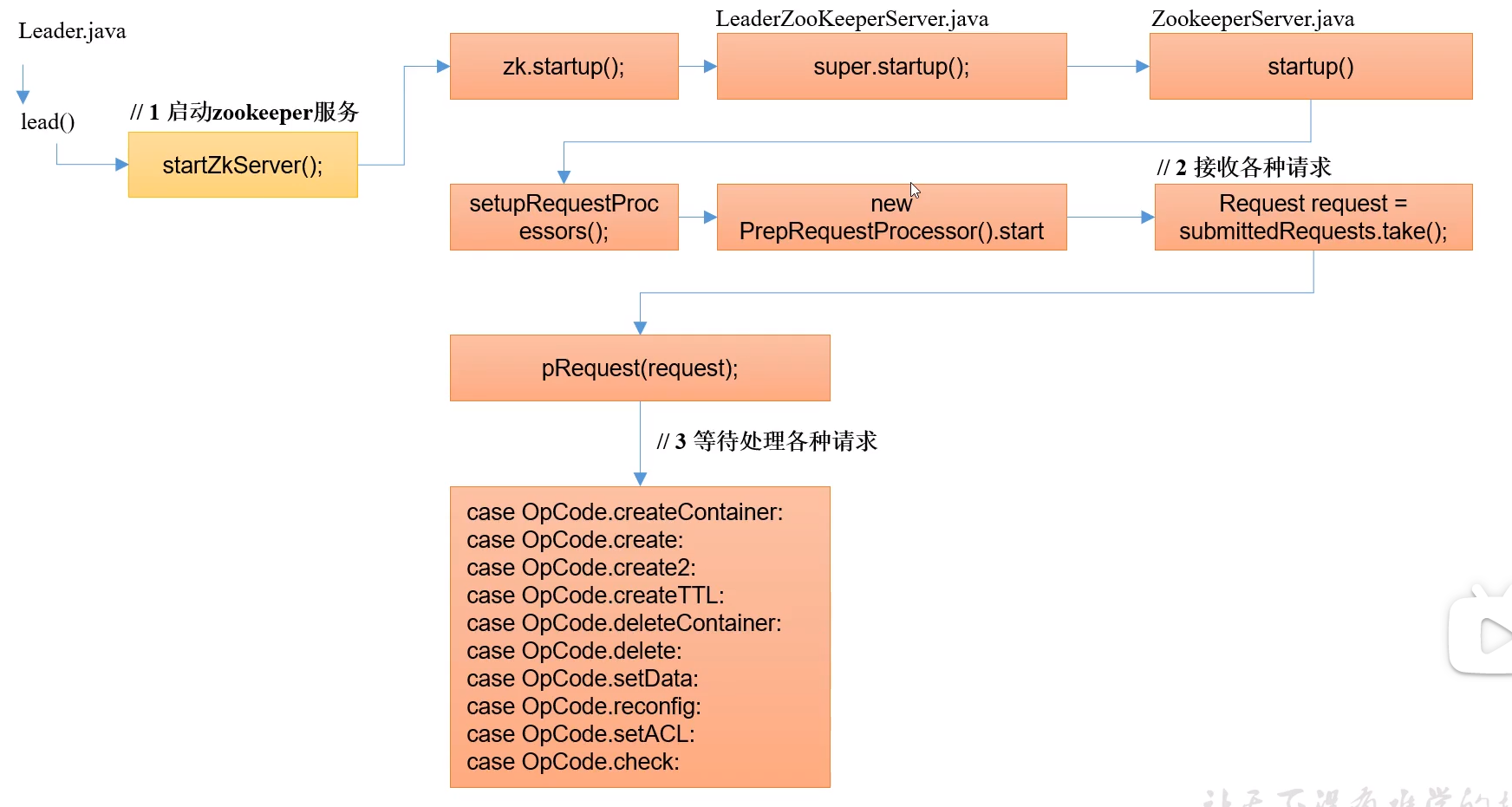
6、leader 发送提交确认。

7、leader发送新leader同步确认。Follower返回ack。、

8、过半数选举成功。

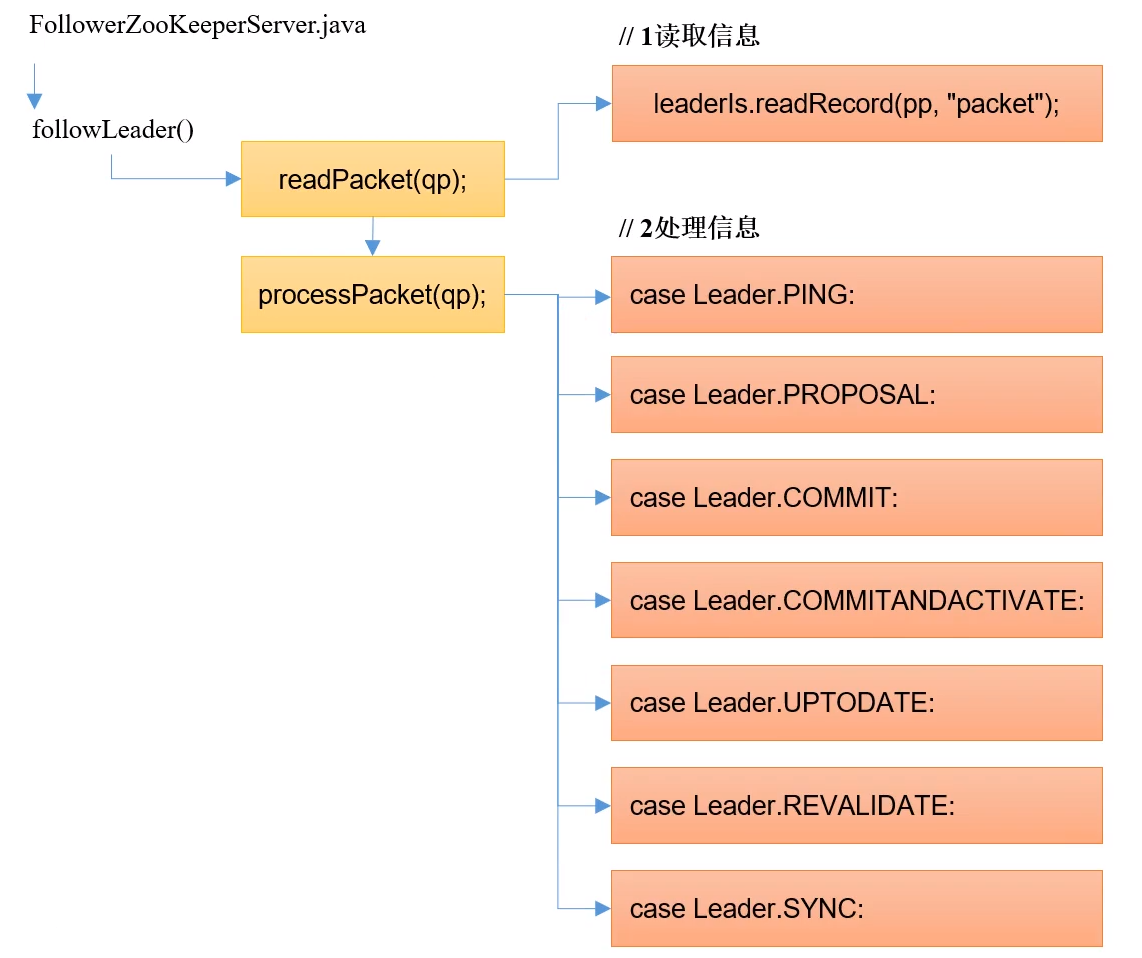
## leader和follower启动后的工作

### leader



启动之后开辟线程去获取socket消息，开始处理接收的消息，根据不同的消息类型去处理。

### Follower



启动之后开辟线程去读取ssocket消息，开始处理接收的消息，根据不同的消息类型去处理。

### 客户端启动工作

# Zookpeeper功能实现

## Master-worker协同

变为master:

M1：Create -e /master “m1:2223”

M2：Create -e /master “m2:2223” 尝试变为master失败

M2: start -w /master 开始监听master结点。

监控woker状态:

Ls -w /workers

Workers:

W1:创建woker1结点 create -e /workers/w1 “w1:2224

W2:创建woker1结点 create -e /workers/w2 “w2:2224

Worker失败之后会通知mater结点。

## 分布式队列

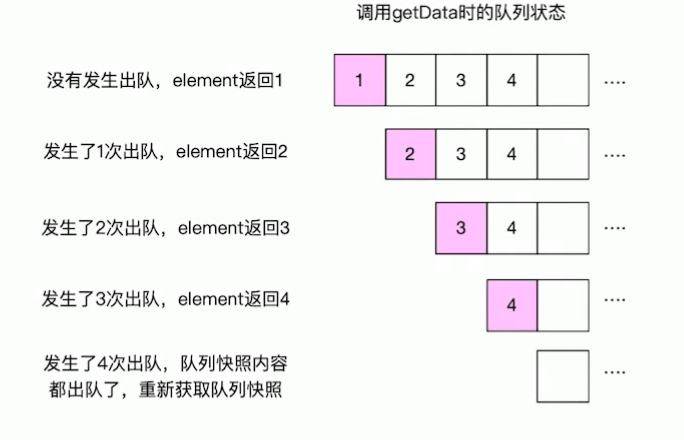
### 队列实现思想

使用路径为/queue的znode下的节点表示队列中的元素。/queue下的节点都是顺序持久化znode。这些znode名字的后缀数字表示了对应队列元素在队列中的位置。Znode名字后缀数字越小,对应队列元素在队列中的位置越靠前。Recipe说明: Queues.

### 方法

offer方法:在/queue下面创建一个顺序znode,因为znode的后缀数字是/queue下面现有znode最大后缀数字加1,所以该znode对应的队列元素处于队尾。

Elements方法：

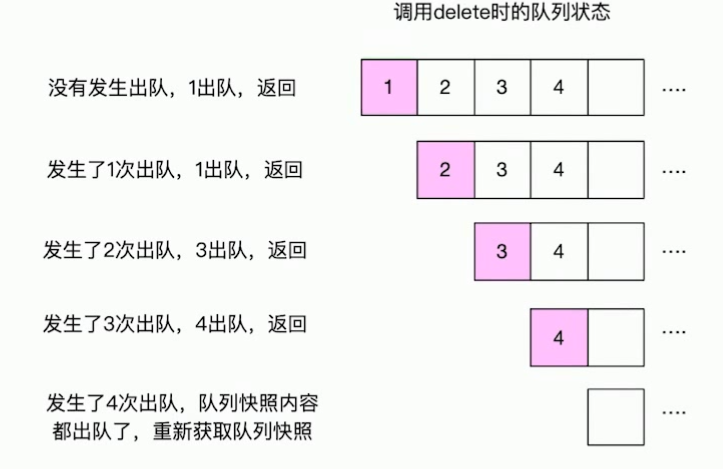


1、throw new NosuchElementException () :因为element方法读取到了队列为空的状态,所以抛出NoSuchElementException是正确的。

2、return zookeeper.getetData (dir+”/”+headNode,false,null):childNames保存的是队列内容的一个快照。这个return语句返回快照中还没出队的后缀数字最小的znode,如果队列快照的元素都出队了,重试。

Remove: take方法和element方法类似。值得注意的是getData的成功执行不意味着

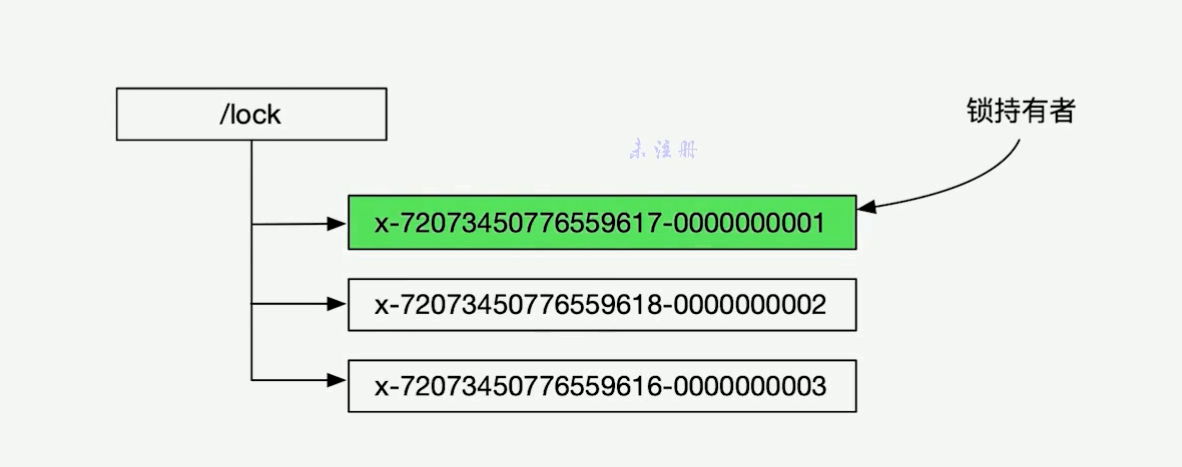
出队成功,原因是该队列元素可能会被其他用户出队。



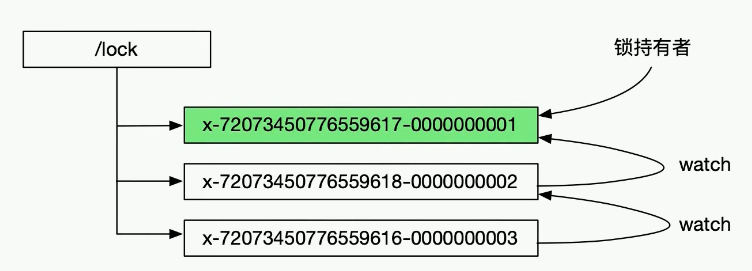
## 分布式锁

### 设计原理

使用临时顺序znode来表示获取锁的请求,创建最小后缀数字znode的用户成功拿到锁。



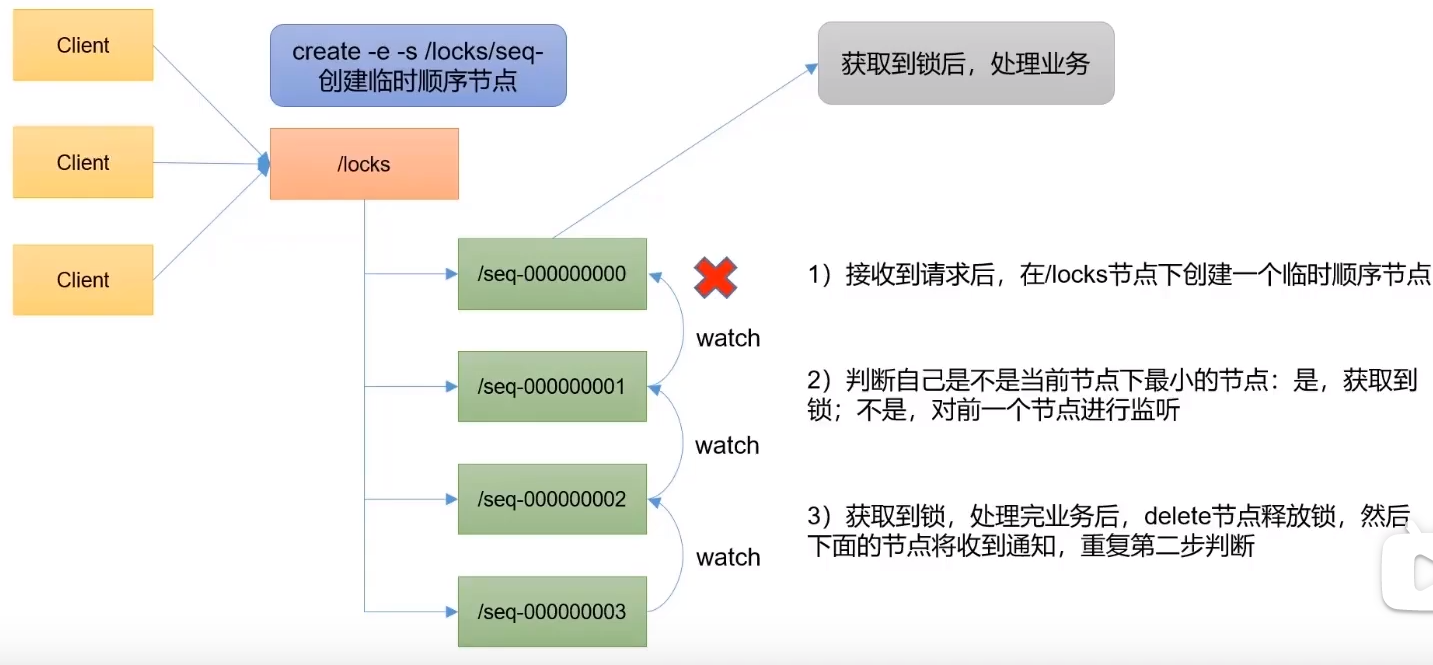
### 分布式锁的羊群效应



把锁请求者按照后缀数字进行排队,后缀数字小的锁请求者先获取锁。如果所有的锁请求者都watch锁持有者,当代表锁请求者的znode被删除以后,所有的锁请求者都会通知到,但是只有一个锁请求者能拿到锁。这就是羊群效应。

为了避免羊群效应,每个锁请求者watch它前面的锁请求者。每次锁被释放,只会有一个锁请求者会被通知到。这样做还让锁的分配具有公平性,锁定的分配遵循先到先得的原则。

### 分布式原理



1、zk服务启动

启动时去判断lock结点(持久化结点)是否存在，如果不存在去创建结点。

2、lock

1、在lock下创建子节点，该结点为临时的序列结点。

"/locks/" + "seq-"+zk帮我们生成的序号

2、获取当前lock下的所有结点，判断创建的节点是否是最小的序号节点。

如果是获取到锁。

如果不是，对当前结点的前一个结点注册一个watch。

3、对前一个结点注册wathch。

1、创建的当前结点例如: // 获取节点名称 seq-00000005

2、获取所有的结点进行排序。

3、从list中获取到seq-00000005在的位置。

4、然后注册的watch在seq-00000004位置。

5、使当前结点await，可以使用countdown。

7、获取到监听事件后countdown.count();

3、unlock

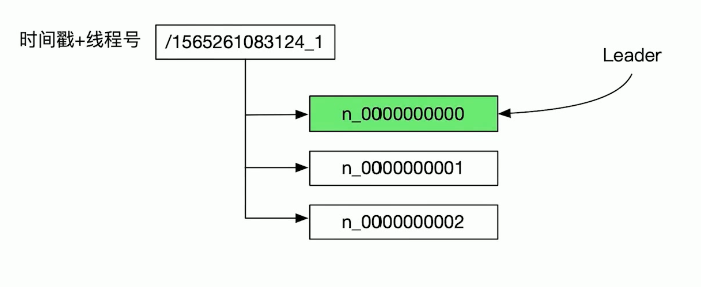
1、客户端直接将该结点删除。

2、curator 实现分布式锁。科瑞特。

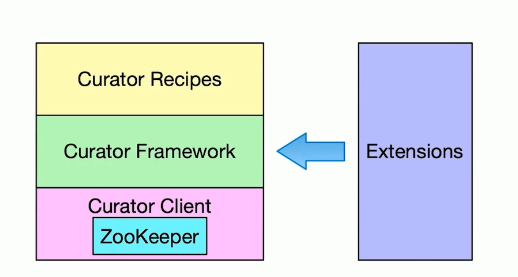
## 分布式选举

### 设计思想

使用临时顺序znode来表示选举请求,创建最小后缀数字znode的选举请求成功。在协同设计上和分布式锁是一样的,不同之处在于具体实现。不同于分布式锁,选举的具体实现对选举的各个阶段做了监控。.



### Curator实现



Client :封装了Zookeeper类,管理和Zookeeper集群的连接,并提供了重建连接机制。

Framework:为所有的ZooKeeper操作提供了重试机制,对外提供了一个Fluent风格的API

Recipes:使用framework实现了大量的

ZooKeeper协同服务。

Extensions :扩展模块。

### 选举原理

## 服务发现

### 服务发现

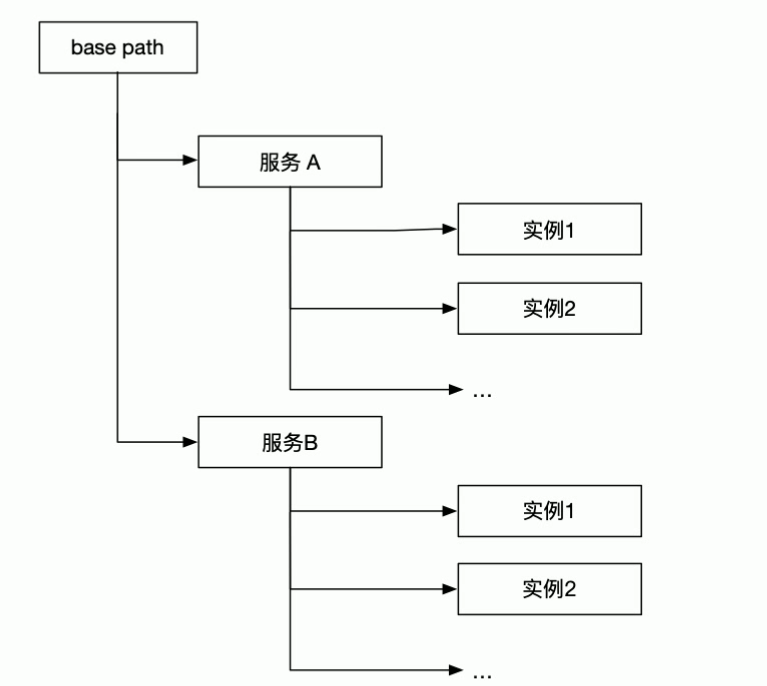
服务发现主要应用于微服务架构和分布式架构场景下。在这些场景下,一个服务通常需要松耦合的多个组件的协同才能完成。服务发现就是让组件发现相关的组件。服务发现要提供的功能有以下3点:

服务注册。

服务实例的获取。

服务变化的通知机制。

### curator-x-discovery设计



使用一个base path作为整个服务发现的根目录。在这个根目录下是各个服务的的目录。服务目录下面是服务实例。实例是服务实例的JSON序列化数据。服务实例对应的znode节点可以根据需要设置成持久性、临时性和顺序性。